agena >>

a programming language

**primer and reference
for version 2.36.1**
by alexander walz

January 29, 2023

Credits

The Sources

Agena has been developed on the ANSI C sources of Lua 5.1, written by Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. Used by their kind permission back in 2006.

Chapter 7: Standard Library documentation

A substantial portion of Chapter 8 has been taken from the Lua 5.1 Reference Manual written by Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. Used by kind permission.

environ.anames

environ.anames has been invented by Joe Riel, put to the Maple community back in the early nineties.

case of statement

The original code has been written by Andreas Falkenhahn and posted to the Lua mailing list on September 01, 2004. In Agena, the functionality has been extended to check multiple values in the **of** branches.

skip statement

The **skip** functionality for loops has been written by Wolfgang Oertl and posted to the Lua Mailing List on September 12, 2005.

environ.globals base library function

The original Lua and C code for **environ.globals** has been written by David Manura for Lua 5.1 in 2008 and published on www.lua.org. The C source has been changed so that in Agena, C functions are no longer checked.

mkdir, chdir, and rmdir functions in the os library

These functions are based on code taken from the `lposix.c` file of the POSIX library written by Luiz Henrique de Figueiredo for Lua 5.0. These functions are themselves based on the original ones written by Claudio Terra for Lua 3.x.

No automatic auto-conversion of strings to numbers

was inspired by Thomas Reuben's no_auto_conversion.patch available at lua.org.

Kilobyte/Megabyte Number Suffix ('k', 'm')

taken from Eric Tetz's k-m-number-suffix.patch available at lua.org.

Binary and octal numbers ('0b', '0o')

taken from John Hind's Lua 5.1.4 patch available at lua.org.

Integer division

taken from Thierry Grellier's newluaoperators.patch available at lua.org.

math.fraction

was originally written in ANSI C by Robert J. Craig, AT&T Bell Laboratories.

The math library functions **eps**, **epsilon**, **exponent**, **issubnormal**, **mantissa**, **math.frexp**, **math.nextafter**, **math.wrap**, **modf**, **round**, **zerosubnormal**, **cis**, **math.sincos**, **arctan**, **arctan2**, **sin**, **cos**, **+++** and **---** operators

use a modified versions of C functions that have originally been published by Sun Microsystems with the fdlibm IEEE 754 floating-point C library. See Appendix B3 for the licence.

calc.diff

based on Conte and de Boor's `Coefficients of Newton form of polynomial of degree 3`.

Advanced precision algorithm used in **for/to loops**, **sumup**, **calc.fsum**, **linalg.trace**, **stats.amean**, **factory.count**, **stats.cumsum**, and **stats.sumdata**.

The method to prevent round-off errors in iterations with non-integral step sizes has been developed by William Kahan and published in his paper `Further remarks on reducing truncation errors` as of January 1965. Agena in some cases uses a modified version of the Kahan algorithm developed by Kazufumi Ozawa, published in his paper `Analysis and Improvement of Kahan's

Summation Algorithm`. Especially the statistics function use the Kahan-Babuška variant described by Andreas Klein in his study `A generalized Kahan-Babuška-Summation-Algorithm`.

calc.minimum, calc.maximum

use the subroutine **calc.fminbr** originally written by Dr. Oleg Keselyov in ANSI C which implements an algorithm published by G. Forsythe, M. Malcolm, and C. Moler, `Computer methods for mathematical computations`, M., Mir, 1980, page 202 of the Russian edition.

bernoulli, besselj, bessely, euler, lambda

are completely or largely based on the functions originally written in FORTRAN by Shanjie Zhang and Jianming Jin, Computation of Special Functions, Copyright 1996 by John Wiley & Sons, Inc. Used by Jianming Jin's kind permission.

Graphics

The graphical capabilities of Agena in the Solaris, Linux, Mac, and Windows versions have been made possible through a Lua binding of Alexandre Erwin Ittner to the g2 graphical library which has been written by Ljubomir Milanovic and Horst Wagner.

ADS package

The core ANSI C functions to create, insert, delete and close the database have been written by Dr. F. H. Toor.

MAPM binding

Mike's Arbitrary Precision Math Library has been written by Michael C. Ring. See Appendix B6 for the licence.

The MAPM Agena binding is an adaptation of the Lua binding written by Luiz Henrique de Figueiredo, put to the public domain.

Year 2038 fix for 32-bit machines

was written by Michael G. Schwern, and has been published under the MIT licence at http://github.com/schwern/y2038.

gzip package

and its description of the binding has originally been written and published under the MIT licence by Tiago Dionizio for Lua 5.0.

Internal string concatenation

Some internal initialisation routines use a C function written by Solar Designer placed in the public domain.

Functions **arctan**, **expx2**, **gamma**, **lngamma**, **calc.Ai**, **calc.Bi**, **calc.dawson**, **calc.dilog**, **calc.Ci**, **calc.Chi**, **calc.En**, **calc.fresnelc**, **calc.fresnels**, **calc.ibeta**, **calc.igamma**, **calc.igammc**, **calc.invibeta**, **calc.polylog**, **calc.Psi**, **calc.Si**, **calc.Shi**, **calc.Ssi**, **calc.zeta**, **stats.gammad**, **stats.gammadc**, and **stats.invnormald**

use algorithms written in ANSI C by Stephen L. Moshier for the Cephes Math Library Release 2.8 as of June, 2000. Copyright by Stephen L. Moshier.

erf, erfc, inverf, inverfc, calc.intde, calc.intdei, calc.intdeo

These functions use procedures originally written in C by Takuya Ooura, Kyoto, Copyright(C) 1996 Takuya OOURA: "You may use, copy, modify this code for any purpose and without fee."

math.random

The algorithm used to compute random numbers has been written by George Marsaglia and published on en.wikipedia.org.

io.anykey

The Linux version uses code written by Johnathon in 2008 which was published under the MIT licence.

xBASE file support

The **xbase** package is a binding to xBASE functions written by Frank Warmerdam in ANSI C for the Shapelib 1.2.10 and 1.3.0 libraries. The Shapelib library has been published under the MIT licence.

The net package

Most of the functions are based on Jürgen Wolf's C examples published in his book `C von A bis Z`, 3rd Edition, Galileo Computing, Bonn, 2009.

`Beej's Guide to Network Programming, Using Internet Sockets`, written by Brian "Beej Jorgensen" Hall, was of great help. Some of the **net** functions use part of Mr. Hall's public domain code published in his tutorial. Copyright © 2009 Brian "Beej Jorgensen" Hall.

Studying the code of the LuaSocket 2.0.2 package, Copyright © 2004-2007 by Diego Nehab, and published under the MIT licence, was very worthwhile.

strings.dleven

The implementation of Damerau-Levenshtein Distance is a blend of C code written by Lorenz and Anders Sewerin Johansen.

utils.readxml

The original version of the core XML parser has been written in Lua 5.1 by Roberto Ierusalimschy, published on LuaWiki.

utils.decodeb64 and **utils.encodeb64**

The Base64 functions have been originally written in pure ANSI C by Bob Trower, Copyright (c) 2001, published under the MIT licence.

printf

was taken from the compat.lua file shipped with the Lua 5.1 sources published under the MIT licence.

.. operator and **{} indexing**

are based on code written by Sven Olsen, published in Lua Wiki/Power Patches.

copy

The deep copying mechanism has originally been written by Kurt Jung and by Aaron Brown for Lua, and published in their book 'Beginning Lua Programming', Wiley Publishing, Indianapolis, Indiana, 2007, page 151.

os.getenv, **os.setenv**, **os.environ**

have been written by Mark Edgar, Copyright 2007, published under the MIT licence, and were taken from http://lua-ex-api.googlecode.com/svn.

bags package

The idea and its core implementation - ported to C - has been taken from the book `Programming in Lua` by Roberto Ierusalimschy, 2nd Edition, Lua.org, p. 102.

xml package

The xml package actually is the LuaExpat binding to the expat library with some few Agena-specific non-OOP modifications. LuaExpat 1.0 was designed by Roberto Ierusalimschy, André Carregal and Tomás Guisasola as part of the Kepler Project which holds its copyright. The implementation was coded by Roberto Ierusalimschy, based on a previous design by Jay Carlson.

LuaExpat development was sponsored by Fábrica Digital and FINEP.

bintersect, **bminus**, **bisequal**, **stats.obcount**

The algorithm for binary comparison has been taken from Niklaus Wirth's book, `Algorithmen und Datenstrukturen mit Modula-2`, 4th ed., 1986, p. 58.

linalg.mulrow, **linalg.mulrowadd**, **stats.deltalist**, **stats.cumsum**, **stats.colnorm**, **stats.rownorm**, **stats.sumdata**

These functions have been inspired by the deltaList, cumulativeSum, centralDiff, colNorm, rowNorm, mrow, and mrowdd functions available on the TI-Nspire™ CX CAS.

linalg.scale, **stats.scale**

is a port of function REASCL, included in the ALGOL 60 NUMAL package published by The Stichting Centrum Wiskunde & Informatica (Stichting CWI) (legal successor of Stichting Mathematisch Centrum) at Amsterdam. Original authors: T. J. Dekker, W. Hoffmann; contributors: W. Hoffmann, S. P. N. van Kampen.

os.now

uses C routines of the IAU Standards of Fundamental Astronomy (SOFA) Libraries, See Appendix B5 for the licence.

Functions **calc.clampedspline**, **calc.clampedsplinecoeffs**, **calc.interp**, **calc.neville**, **calc.newtoncoeffs**, **calc.nokspline**, **calc.noksplinecoeffs**

use C++ routines (ported to C) provided or written by Professor Brian Bradie, Department of Mathematics, Christopher Newport University, VA, to the course `An Introduction to Numerical Analysis with Applications to the Physical, Natural and Social Sciences`. There have been no copyright remarks, so at least Agena's MIT licence is *not* applicable to the source files `interp.c` and `interp.h`.

stats.smallest

is based on N. Devillard's C implementation of an algorithm published in various books written by Niklaus Wirth, published for example in `Algorithmen und Datenstrukturen mit Modula-2`. Mr. Devillard put his code in the public domain.

strings.isiso* and **strings.iso*** functions

use ISO 8859/1 Latin-1 bit vector tables taken from the entropy utility ENT written by John Walker, January 28th, 2008, Fourmilab, put in the public domain.

astro.moonriseset

Uses C functions Copyright © 2010 Guido Trentalancia IZ6RDB. This program is freeware - however, it is provided as is, without any warranty.

astro.phase

Uses C functions taken from: http://www.voidware.com/moon_phase.htm. There have not been any copyright remarks.

astro.sunriseset

Uses C functions written as DAYLEN.C, 1989-08-16. Modified to SUNRISET.C, 1992-12-01, (c) Paul Schlyter, 1989, 1992. Released to the public domain by Paul Schlyter, December 1992.

astro.cdate & **astro.jdate**

uses C routines of the IAU Standards of Fundamental Astronomy (SOFA) Libraries,
See Appendix B5 for the licence.

strings.utf8size

of the core C code procedure has been written by mpez0, published at
StackOverflow.

strings.isutf8

of the core C code procedure has been written by written by Christoph,
published on StackOverflow.

strings.isotolatin & **strings.isotoutf8**

of the core C code procedures have been written by Nominal Animal published
on StackOverflow.

strings.glob

uses C code written by Arjan Kenter, Copyright 1995, Arjan Kenter.

stats.sorted

uses an iterative Quicksort algorithm written by Nicolas Devillard in 1998, put to
the public domain.

/%, *%, +%, -%, %% operators, **math.dd**, **math.dms**, **math.splitdms**, **polar**,
stats.cdf, **stats.numbcomb**, **stats.numbperm**, and **stats.pdf**

have been inspired by the TI™-30 ECO RS, TI™-30X Pro, Sharp™ EL-W531XG and
HP 35s pocket calculators.

E, **Exp**

as a constant, defines the former Maple V Release 3 implementation of E =
exp(1) = 2.71828182845904523536.

Complex arithmetic

for various mathematical functions and operators has been implemented by primarily using Maple V Release 3, Maple V Release 4, and Maple 7.

io.getclip and io.putclip

are based on C code written by banders7, published on Daniweb.

try/catch statement

has been invented and written by Hu Qiwei for Lua 5.1 back in 2008, and has been extended for Agena.

debug.getinfo

the 'a'/arity extension has been written by Rob Hoelz in 2012.

calc.polyfit & calc.linterp

uses C code published by Harika in 2013 at http://programbank4u.blogspot.de.

Review of the Agena interpreter at the Web

Many thanks to **softpedia.com** for the very kind critique and fine ranking.

linalg.det & linalg.inverse

are based on C functions written by Edward Popko published on Paul Bourke's website at http://paulbourke.net/miscellaneous.

redo & relaunch

have been inspired by the Ruby programming language.

linalg.gsolve

is based on C functions written by Edward Popko and Alexander Evans; for the former see the link above, and for the latter the following address: http://www.dailyfreecode.com/code/basic-gauss-elimination-method-gauss-29 49.aspx.

calc.simaptive and **linalg.ludecomp**

are based on C functions written by RLH, available at
http://www.mymathlib.com, Copyright © 2004 RLH. All rights reserved.

~=, **~<>**, **approx**, **qmdev**

use methods developed by Donald Knuth.

calc.Ei

uses a combination of C algorithms written by Stephen L. Moshier and RLH.

linalg.rref

is based on a C# function published at http://rosettacode.org.

linalg.forsub

is based on an algorithm explained by Timothy Vismor found on his site
http://vismor.com.

cordic package

is based on a C package written by John Burkardt, taken from
http://people.sc.fsu.edu/~jburkardt/c_src/cordic/cordic.c, with modifications
using Maple V Release 4 and TI-Nspire CX CAS. Sources provided separately.

libusb binding

is based on lualibusb1 - Lua binding for libusb 1.0, written by Tom N Harris. See:
http://lualibusb1.googlecode.com.

stats.extrema

is the Agena port of the `peakdet` function written by Eli Billauer for MATLAB.

mdf, **xdf**

have been inspired by the Sharp PC-1403H pocket computer.

os.cpuload, **os.drivestat, os.getenv, os.realpath** & **os.setenv**

are based mainly on procedures taken from Nodir Temirkhodjaev's LuaSys package.

utils.readini

uses modified C sources written by Nicolas Devillard for his iniparser 3.1 package.

Various OS/2 operating system functions

have been made possible by the website http://www.edm2.com/os2api.

llist and heaps packages

The C implementation of singly and doubly-linked lists and AVL trees has been accomplished by reading Michal Kottman's tip at nabble.com on how to code new data structures using Lua's userdata and how to anchor values into the registry. The algorithms themselves have originally been written in C by Martin Broadhurst.

stats.dbscan & **stats.neighbours**

The dbscan algorithm has been invented by Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu, published at University of Munich. The Agena port is based on a Matlab implementation written by Peter Kovesi, Centre for Exploration Targeting, The University of Western Australia, with **stats.neighbours** a C-based split-off.

hashes package

uses code published by RSA Data Security, Inc. Copyright (C) 1990. All rights reserved. For further credits, please see the hashes.c file in the Agena sources.

math.ceilpow2 and **math.ilog10**

use code presented by Sean Eron Anderson at his `Bit Twiddling Hacks` webpage http://graphics.stanford.edu/~seander/bithacks.html.

os.cdrom, **os.ismounted**, **os.isremovable**, **os.isvaliddrive**

The Windows versions are based on code published at MSDN, page http://support.microsoft.com/kb/165721#. The Linux version of **os.cdrom** is based on Jürgen Wolf's C book `C von A bis Z`, 3rd Edition, Galileo Computing, Bonn, 2009. The OS/2 version of **os.cdrom** is based on code found on the OS/2 Hobbes FTP server at NMSU, left without any copyright remarks.

os.terminate

The OS/2 version is largely based on Mark Kimes' public domain implementation.

os.monitor

The Linux version is based on Dave Drager+'s recommendation published at his blog.

hypot2 and antilog$_n$ operators

have been inspired by the Sinclair Scientific Programmable pocket calculator.

math.eps, **stats.isall**, **stats.isany**, and **linalg.reshape** functions

have been inspired by Matlab.

stats.gmean

uses an algorithm taken from the COLT sources published by CERN**, Geneva.**

gdi.plotfn

has been improved by Slobodan from Serbia.

oftype metamethod

to check structures at function invocation has been proposed by Slobodan from Serbia.

stats.durbinwatson, **stats.standardise**, and **stats.sumdataln**

have been inspired by the COLT package published by CERN, Geneva.

<<<< and >>>> operators, bytes.arshift32, bytes.extract32, bytes.replace32

have been implemented using Lua 5.2 and 5.3 code and Rupert Tombs' arithmetic right-shift implementation.

Chapter 6.24

is based on examples published at http://www.lua.org/pil/16.html.

Exit and restart handling

via **environ.onexit** has been inspired by MuPAD 2.5.

with and related statements

are based on a Lua 5.1 power patch written by Peter Shook (`Unpack Tables by Name`).

math.dms

uses an algorithms proposed by user807566 on StackOverflow.

case of *boolean condition* variant

has been inspired by the Go programming language.

Numeric ranges in case/of clauses

have been inspired by the Fortran 90 programming language.

math.fma

for those platforms that do not provide a built-in fma C function, is based on a method proposed by Z boson on StackOverflow.

math.signbit

for those platforms that do not provide a built-in signbit C function, is based on a Sun Microsystems implementation.

math.signbit

Its original version has been written by Jacob Rus for Lua, taken from: https://gist.github.com/jrus/3197011.

math.wrap

Is based on Tim Cas' answer #4633177 on StackOverflow and the restrictsymm function of the Julia programming language.

Sinclair ZX Spectrum package

clones Spectrum ROM Z80 assembler routines disassembled by Dr. Ian Logan and Dr. Frank O'Hara.

math.eps

optionally uses a formula suggested by trashgod on StackOverflow to compute a small epsilon value that is suited for mathematical C double operations.

dBASE version numbers

printed in the description of **xbase.attrib** have been taken from: http://stackoverflow.com/questions/3391525, answered by Les Paul.

round, mdf, and xdf

use an underlying C routine posted by Larry I Smith, see: https://bytes.com/topic/c/answers/521405-rounding-nearest-nth-digits.

math.cld, math.fld, math.flipsign, math.isqrt, math.lnfact, and math.powmod

have been ported from or have been inspired by the corresponding functions written in the Julia programming language, published under the MIT licence.

strings.appendmissing, **strings.between**, **strings.chop**, **strings.chomp**, **strings.con-tains**, **strings.uncapitalise**, **strings.iswrapped**, **strings.wrap**, and **strings.wrapmissing**

are ports of StringUtils functions part of the Apache Commons Lang 3.5 API.

astro.hdate and **os.date** ('*sdn' format)

use C functions written by Scott E. Lee, see http://www.rosettacalendar.com.

hashes.mix64 and **hashes.mix64to32**

use Thomas Wang's C procedures, taken from gist.github/badboy/6267743.

times

is based on the corresponding Haskell function iterate.

for/until loops

have been inspired by COBOL.

math.sincos

uses Elliot Saba's sincos implementation.

math.accu

uses Julia Language's Kahan-Babuška-Neumaier compensated summation.

hashes.droot, **hashes.parity**, **hashes.reflect**

use Henry S. Warren's code published with his book `Hacker's Delight`.

hashes.pjw, **hashes.rs**, **hashes.bp**

are based on C functions written by Arash Partow.

map/@ extension to support function composition & **reduce**

> have been inspired by Slobodan's feedback and an excellent introduction to functional programming written by Mary Rose Cook.

bloom filter plus package

> is based on C code created by Simon Howard, see Appendix B9 for ISC licence.

factory plus package

> has been inspired by the `functools` package in Python 3.

strings.a64 and **hashes.sha512**

> use C code from the musl-1.1.19 library, MIT licence.

? statement, **prepend**, **linalg.iszero**, **linalg.isallones**, thus indirectly **satisfy**

> have been inspired by the Axiom Computer Algebra System.

getorset

> has been inspired by the `getOrElseUpdate` operator in the Scala programming language.

if is operator and compound assignments, **+:=**, **-:=**, etc.

> have been inspired by Algol 68.

bytes.pack, **bytes.packsize**, **bytes.unpack**, **tables.move**, and the **utf8** package
> have been taken from Lua 5.3.5 or Lua 5.4.0 RC 4 (**utf8**, **move**).

GMP 6.1.2 port for OS/2
> compiled by KO Myung-Hun has been used to compile the **mp** binding.

dual package
> uses definitions primarily found at blog.demofox.org and adl.stanford.edu.

os.iterate
> has been derived from listing published in `Programming in Lua` 2nd Ed.,
> pp 271f., by Roberto Ierusalimschy.

com package
> is largely based on the LuaSys package v1.8, written by Nodir Temirkhodjaev.

assignments in conditions of while loops, if and case of statements
> were inspired by Icon and C.

duplicate parser warnings for duplicate local variable declaration
> have originally been designed by Domingo Alvarez Duarte for Lua 5.1.

shift
> has been written by StackOverflow user ryanpattison for Lua.

type anything and more or less **constants**
> have been inspired by Maple.

erfcx, **calc.scaleddawson**, **calc.w**
> use code written by Steven G. Johnson, October 2012, MIT licence.

os.netuse, **os.netsend** & **os.netdomain**
> use code written by Antonio Escaño Scuri for the NTLua 3.0 package,
> MIT licence.

rings package
> has been designed by Roberto Ierusalimschy and Tomás Guisasola for Lua 5
> as part of the Kepler Project. The implementation was originally coded by
> Tomás Guisasola. Rings development was sponsored by Fábrica Digital.
> MIT licence.

utils.decodeb85 and **utils.encodeb85**

The Base85 functions have been originally written in C by Rafa Garcia, Copyright
(c) 2016-2018, published under the MIT licence.

utils.decodea85 and **utils.encodea85**

The ASCII85 conversion functions have been written in C by Luiz Henrique de Figueiredo, placed in the public domain.

strings.pack, **strings.packsize** and **strings.unpack**

have been taken from Lua 5.4.4, Lua.org, PUC-Rio, MIT licence.

bimaps package
has originally been written by Pierre 'catwell' Chapuis for Lua.
Copyright (C) 2013-2015 by Pierre Chapuis. MIT licence.

heaps package
is based on a Lua package written by Geoff Leyland, New Zealand.
Copyright (c) 2008-2011 Incremental IP Limited. MIT licence.

aconv package
is based on the Lua-iconv 7 package for Lua 5.1, 2005 - 2011, MIT licence, written by Alexandre Erwin Ittner.

.

Finally, due to very kind help and feedback, in chronological order

Many thanks to the Lua team at PUC-Rio, Brazil, and to Agena users in Israel, Italy, Australia, Palestine, Poland, Serbia, the OS/2 community, and to users of other nations.

Table of Contents

Part **One**

Primer

Chapter **One**

Introduction

1 Introduction

1.1 Abstract

Agena is a procedural programming language designed to be used in scientific, educational, linguistic, and many other applications, including scripting.

Agena provides real and complex arithmetic, graphics, efficient text processing, flexible data structures, intelligent procedures, package management, plus various multi-user configuration facilities.

Its syntax looks like very simplified Algol 68 with elements taken primarily from Maple, Lua and SQL. It has been implemented on the ANSI C sources of Lua 5.1 created by Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes.

Agena binaries are available for Solaris, Linux, Windows, OS/2, Mac OS X, Raspberry Pi, and DOS.

You may download Agena, its sources, and its manual from

<div align="center">http://sourceforge.net/projects/agena.</div>

1.2 Features

Agena combines features of Lua 5, Maple, Algol 60, Algol 68, ABC, SQL, ANSI C and BASIC.

Agena provides all the common functionality found in imperative languages:

- statements,
- loops,
- conditions,
- procedures.

It also has extended programming features described later in this manual, such as:

- high-speed processing of extended data structures,
- fast string and mathematical operators,
- extended conditionals,
- abridged and extended syntax for loops,
- special variable increment, decrement and deletion statements,
- efficient recursion techniques,
- arbitrary precision mathematical libraries,
- a network package to exchange data over the Internet and LANs,
- easy-to-use package handling,
- and much more.

Like Lua, Agena is untyped and includes the following basic data structures: numbers, strings, booleans, tables, and procedures. In addition to these types, it

also supports Cantor sets, sequences, registers, pairs, complex numbers, linked lists, and multisets. With all of these types, you can build applications easily.

1.3 In Detail

Agena offers various flow control facilities such as

- **if/elif/else** conditions,
- **case of/else** conditions similar to C's switch/case statements,
- **if** operator to return alternative values,
- numerical **for/from/to/downto/by** loops with optional start, stop and step values, and automatic round-off error correction of iteration variables,
- combined **for/while** and **for/until** loops,
- **for/in** loops over strings and complex data structures,
- **while** and **do/as** loops similar to Modula's while and repeat/until iterators,
- **do/od** loops equal to the ones in Maple,
- a **skip** statement to prematurely trigger the next iteration of a loop,
- a **break** statement to prematurely leave a loop,
- a **do nothing** statement which does not do anything,
- fast and easy data type validation with the optional double colon facility in parameter lists.

Data types provided are:

- rational and complex numbers with extensions such as **infinity** and **undefined**,
- strings,
- booleans such as **true**, **false**, and **fail**,
- the **null** value depicting the absence of a value,
- multipurpose tables implemented as associative arrays to hold any kind of data, taken from Lua,
- Cantor sets as collections of unique items,
- sequences and registers, i.e. vectors, to internally store items in strict sequential order,
- pairs to hold two values or pass options to procedures,
- threads, userdata, and lightuserdata inherited from Lua.

For performance, most basic operations on these types have been built into the Agena kernel.

Procedures with full lexical scoping are supported, as well, and provide the following extensions:

- the << (args) -> expression >> syntax to easily define simple functions,
- user-defined types for procedures to allow individual handling,
- uder-defined types for tables, sets, sequences, registers and pairs,
- a facility to return predefined results,
- remember tables for high-speed recursion,
- closures which let functions remember their state, taken from Lua,

- the **nargs** system variable which holds the number of arguments actually passed to a procedure,
- metamethods to define operations for tables, sets, sequences, registers and pairs, inherited from Lua,
- OOP-style methods for tables,
- self-defined binary operators.

Some other features are:

- graphics in the Solaris, Mac, 32-bit Linux, Raspberry Pi, and Windows editions, provided by the **gdi** package,
- IPv4 networking with the Internet and LANs,
- functions to support fast text processing,
- configuration of user's environment via the Agena initialisation file,
- an easy-to-use package system also providing a means to both load a library and define short names for all package procedures at a stroke,
- the **binio** package to easily write and read files in binary mode,
- facility to store any data to a file and read it back later,
- undergraduate Calculus, Linear Algebra, and Statistics packages,
- enumeration and multiple assignment,
- transfer of the last iteration value of a numeric **for** loop to its surrounding block,
- scope control via the **scope**/**epocs** keywords,
- efficient stack programming facilities,
- bitwise operators,
- direct access to the file system,
- arbitrary precision mathematical libraries,
- dBASE, XML, CSV, INI, GZIP and TAR file support.

Agena includes all the packages that are part of Lua 5.1. Some of the very basic Lua library functions have been transformed to Agena operators to speed up execution of programmes. The Lua mathematical and string handling packages have been tuned and extended with new features.

Agena code is not compatible to Lua. Its C API, however, has been left unchanged and many new API functions have been added. As such, you can integrate any C package you have already written for Lua by just replacing the Lua- specific header files, see Chapter 17.

1.4 History

I have been dreaming of creating my own programming language for the last 35 years, with my first rather unsuccessful attempt on a Sinclair ZX Spectrum in the early 1980s.

Plans became concrete in 2005 when I learned Lua to write procedures for phonetic analysis and also learned ANSI C to transfer them into a C package. In autumn 2006 the first modifications of the Lua parser started with extensive modifications and extensions of the lexer, parser and the Lua Virtual Machine in summer 2007. Most of Agena's basic functionality had been completed in March

2008, followed by the first new data structure, Cantor sets, one month later, some more data structures, and a lot of fine-tuning and testing thereafter. Finally, in January 2009, the first release of Agena was published at Sourceforge.

Study of many books and websites on various programming languages such as Algol 68, Maple, Algol 60, and ABC, and my various ideas on the `perfect` language helped to conceive a completely new Algol 68-syntax based language with high-speed functionality for arithmetic and text processing.

You may find that at least the goal of designing a perfect language has not been met. For example, the syntax is not always consistent: you will find Algol 68-style elements in most cases, but also ABC/SQL-like syntax for basic operations with structures. The primary reason for this is that sometimes natural language statements are better to reminisce. I have stopped bothering about this inconsistency issue.

After almost four years of development, Agena 1.0 has been released in August 2010.

1.5 Origins

Most of all functionality stems from Lua, Maple and C. Some of my favourite additions to the Lua C sources include:

Maple V Release 3 and later

- **if/elif/else/fi**, **for/while**, **map**, **remove**, **select**, **selectremove**, **subs**, **with**, **readlib**, package management, library.agn, agena.ini, **read**, **save**, substrings, Cantor sets and its operators, sequences, remember tables, **in**, **nargs**, **op**(s), **restart**, **tables.indices**, the **linalg** package, maybe all the pretty printers, argument type checks, **::** type checks, and multiple **::** type parameter checks, surely all mathematical functions and complex arithmetic, and much, much more.

  The Maple language has been designed by Michael B. Monagan, Keith O. Geddes, K. M. Heal, George Labahn, and S. M. Vorkoetter for Waterloo Maple Inc./Maplesoft, Waterloo, Ontario. It is loosely based on Algol 68.

This is also why Agena looks a lot like Maple, and thus somewhat like:

Algol 68

has many times been called the queen of all programming languages and Agena's

- **case/of/esac** control

has originally been introduced with Algol 68.

Algol 60

- **entier.**

Algol 60 is the parent of Algol 68.

Modula-2

- **inc** and **dec**.

C

- **printf**, and most of Lua's system functions,
- compound operators such like c++, etc.

C actually is a descendent of Algol 68.

COBOL

- **for**/**until** loops.

Sinclair ZX Spectrum BASIC

- **clear, cls, int**.

SQL and **ABC**

- **insert**/**into** and thus indirectly **create, delete/from**, and **pop/from**.

PL/I and REXX

- Some of the **strings** library functions have been taken from PL/I and REXX.

Eiffel

- Checking the return type of procedures with the `proc(···) :: <typename>` statement has been taken from this language.

Ada

- inspired the **skip when** and **break when** statements.

Chapter **Two**

Installing & Running Agena

2 Installing and Running Agena

2.1 Sun Solaris 10

In Sun Solaris, and some of its forks, e.g. OpenSolaris, put the gzipped Agena package into any directory. Assuming you want to install the Sparc version, uncompress the package by entering:

```
> gzip -d agena-x.y.z-sol10-sparc-local.gz
```

Then install it with the Solaris package manager:

```
> pkgadd -d agena-x.y.z-sol10-sparc-local
```

This installs the executable into the `/usr/local/bin` folder and the rest of all files into `/usr/agena`. The `/usr/agena/lib` directory is called the `main Agena library folder`.

Make sure you have the *expat, fontconfig, freetype, libg2, libgmp-10, jpeg, libgcc, libgd, libiconv, libintl, libncurses, libmpfr-6, libpng, readline, (lib)xpm,* and *zlib* libraries installed. From the command line, type `agena` and press RETURN.

Image 1: Start-up message in Solaris

The procedure for OpenSolaris and Solaris for x86 CPUs is the same. The package always installs as `SMCagena`.

2.2 Linux

On Debian based x86 distributions, install the 32-bit Stretch deb installer by typing:

```
> sudo dpkg -i --force-all agena-x.y.z-raspi.stretch.i386.deb
```

On Red Hat systems, install the rpm distribution by typing as root:

```
> rpm -ihv --nodeps agena-x.y.z-linux-i386.rpm
```

This installs the executable into the `/usr/local/bin` folder and the rest of all files into `/usr/agena`. The `/usr/agena/lib` directory is called the `main Agena library folder`.

Note that you must have the *expat, fontconfig, freetype, libg2, libgmp-10, libjpeg62, libgcc, libgd* (version 2.0.36 or earlier), *libiconv, libintl, libmpfr-6, libncurses, libpng12, libreadline6, (lib)xpm, x11proto-xext-dev* and *zlib* libraries installed before.

If you have no jpeg library installed on your system, also install *libjpeg62*. **Warning**: overinstalling libjpeg*turbo with *libjpeg62* may totally corrupt your system, as happened on a Raspberry Pi.

From the command line, type `agena` and press RETURN.

The name of the Linux package is `agena`.

2.3 Windows

Just execute the Windows binary installer, and choose the components you want to install.

Make sure you either let the installer automatically set the AGENAPATH environment variable containing the path to the main Agena library folder (the default) or set it later manually in the Windows Control Panel, via the `System` menu.
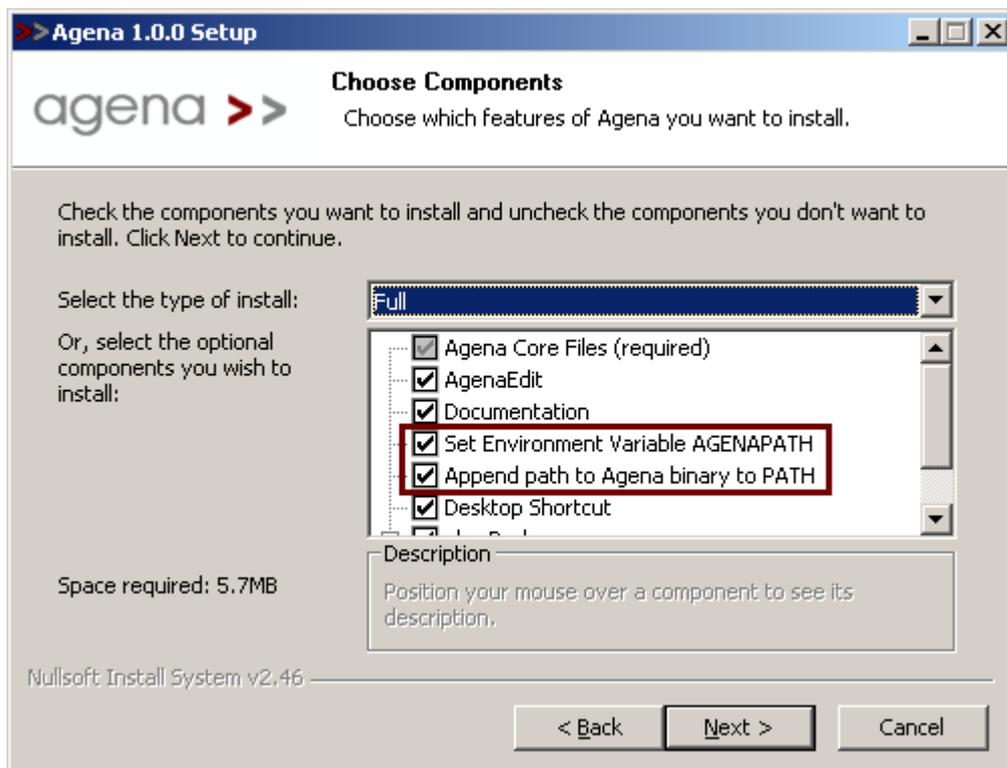
Image 2: Leave the framed settings checked

WARNING: If your system environment variable PATH already consists of 8,000 or more characters, do NOT select the 'Append path to Agena binary to PATH' option, as this might corrupt the PATH setting.

You may start Agena either via the Start Menu, or by typing `agena` in a shell.

Image 3: Start-up message in Windows

If you do not have administrative rights to start the installer, or want to use the interpreter on a removable stick, download the portable version of Agena available at Sourceforge.net and study the readme.w32 file.

For the portable version:

In a NT shell, create a folder called 'agena' anywhere on your drive, change into this directory and decompress the archive into this folder preserving the subdirectory structure of the ZIP file.

Only if you use Windows 2000 or earlier: Now set the environment variable AGENAPATH, referring to the main Agena library `agena.lib` file. For example, if you install Agena into the folder c:\agena, the library files will reside in the c:\agena\lib subfolder, so enter the following statement:

```
set AGENAPATH=c:/agena/lib
```

Note the forward slashes in the path and the system variable name in capital letters.

In XP and later, Agena determines the path to the main Agena library automatically, provided you do not alter the subdirectory structure of the portable distribution.

2.4 OS/2 Warp 4, eComStation and ArcaOS

The WarpIN installer allows you to choose a proper directory for the interpreter, and then installs all files into it.

The dependencies are: WarpIN & kLIBC & ncurses; install using YUM:

```
yum install libc readline ncurses gmp
```

Make sure you either let the installer automatically set the environment variable called AGENAPATH containing the path to the main Agena library folder (the WarpIN default) by leaving the `Modify CONFIG.SYS` entry in the System Configuration window checked, or set it later by manually editing config.sys.

Just enter `agena` in a shell to run the interpreter, or double-click the Agena icon in the programme folder. Agena may require EMX runtime 0.9d fix 4 or higher in OS/2.

2.5 DOS

In DOS, create a folder called `agena` anywhere on your drive, change into this directory and decompress the `agena.zip` file into this folder preserving the subdirectory structure of the ZIP file.

Now set the environment variable `AGENAPATH` in the `autoexec.bat` file. Use a text editor for this. For example, if you installed Agena into the folder `c:\agena`, and the `library.agn` file is in the `lib` subfolder, enter the following line into the `autoexec.bat` file:

```
set AGENAPATH=c:/agena/lib
```

Note the forward slash in the path and the variable name in capital letters.

Also append the path to the `agena` folder to the PATH system variable using backslashes, so that the entry looks something like this:

```
PATH C:\;C:\NWDOS;C:\AGENA\BIN
```

Although it is not necessary in FreeDOS 1.1 or later, at least with Novell DOS 7, you must install `CWSDPMI.EXE` delivered with the DJPGG edition of GCC as a TSR programme before starting Agena. The binary can be found in the DJGPP distribution.

In order to always load this TSR when booting your computer, open the `autoexec.bat` file with a text editor. Assuming the `CWSDPMI.EXE` file is in the `c:\tools` folder, add the following line:

```
loadhigh c:\tools\cwsdpmi.exe -p
```

Novell DOS's command line history works correctly on the Agena prompt.

2.6 Mac OS X 10.5 and above

Simply double-click the `agena-x.y.z-mac.pkg` installer in the file manager and follow the instructions. Do not choose an alternative destination for the package.

The Agena executable is copied into the `/usr/local/bin` folder, supporting files into `/usr/agena`, and the documentation to `/Library/Documentation/Agena`. The `/usr/agena/lib` directory is called the `main Agena library folder`.

Note that you may have to install the *readline* library before.

From the command line, type `agena` and press RETURN.

2.7 Agena Initialisation

When you start Agena, the following actions are taken:

1. The standard packages are initialised so that they become available to the user immediately.

2. All global values are copied from the **_G** table to its copy **_origG**, so that the **restart** function can restore the original environment if invoked.

3. The system variables **libname** and **mainlibname** pointing to the main Agena library folder and optionally to other folders is set by either querying the environment variable AGENAPATH or - if not set - checking whether the current working directory contains the string `/agena` or any other eligible folder name, building the path accordingly.

   The main Agena library folder contains library files with file suffix `agn` written in the Agena language, or binary files with the file suffix `so` or `dll` originally written in ANSI C.

   In UNIX, Mac OS X, and Windows, if the path could not be determined as described before, **libname** and **mainlibname** are by default set to `/usr/agena/lib` in UNIX and Mac OS X, and `%ProgramFiles%\agena\lib` in Windows, if these directories exist and if the user has at least read permissions for the respective folder. The **libname** variable is used extensively by the **import** and **readlib** functions that initialise packages. If it could not be set, many package functions will not be available.

4. Searching all paths in **libname** from left to right, Agena tries to find the standard Agena library `library.agn` and if successful, loads and runs it. The `library.agn` file includes functions written in Agena that complement the C libraries. If the standard Agena library could not be found, a warning message, but no error, is issued. If there are multiple `library.agn` files in your path, only the first one found is initialised.

5. The global Agena initialisation file - if present - with file name `agena.ini` is searched by traversing all paths in **libname** from left to right. As with `library.agn`, this file contains code written in Agena that an administrator may customise with pre-set variables, auxiliary procedures, etc. If the initialisation file does not exist, no error will be issued. If there are multiple Agena initialisation files

in your **libname** path, only the first one found is processed.

In UNIX based systems, the name of the initialisation file may also be `.agenainit`. If both an `.agenainit` and an `agena.ini` file exist, then `.agenainit` is read first.

6. The user's personal Agena initialisation file called `agena.ini` (optionally `.agenainit` in UNIX) - if present - is searched in the user's home folder and run. If this initialisation file does not exist, no error will be issued. After that the Agena session begins. See Appendix A6 for further details.

   In UNIX based systems, if both the `.agenainit` and `agena.ini` files exist, then `.agenainit` is read first.

7. The path to the current user's home directory is assigned to the **environ.homedir** environment variable.

2.8 Installing Library Updates

Sometimes, library updates are provided at Sourceforge if library functions written in the Agena language have been patched or also if new functions written in the language have been developed.

For instructions on how to easily install such an update, have a look at the libupdate.readme file residing on the root of the agena-*x.y.z*-update*n*.zip archive which can be downloaded from the Binaries Agena Sourceforge folder.

In general, the updates can be installed by just unpacking the respective ZIP archive into the main Agena folder.

A library update can be installed on every supported operating system, but you may need administrative rights.

Chapter **Three**

Overview

3 Summary

Let us start by just entering some commands that will be described later in this manual so that you can get acquainted with Agena as fast as possible. In this chapter, you will also learn about some of the basic data types available.

On UNIX-based systems or DOS, type `agena` in a shell to start the interpreter. On OS/2 and Windows, either click the Agena icon in the programme folder or type `agena` in a shell.

3.1 Input Conventions in the Console Edition

Any valid Agena code can be entered at the console with or without a trailing colon or semicolon:

- If an expression is finished with a colon, it is evaluated and its value is printed at the console.

- If the expression ends with a semicolon or neither with a colon nor a semicolon, it is evaluated, but nothing is printed on screen.

You may optionally insert one or more white spaces between operands in your statements.

- To display results in the output window, pass the respective expression to the **print** function, e.g.:

```
print(exp(2*Pi*I)) or a := 1; print(a);
```

You may optionally insert one or more white spaces between operands in your statements.

3.3 Getting Familiar

Assume you would like Agena to add the numbers 1 and 2 and show the result. Then type:

```
> print(1+2)
3
```

If you want to store a value to a variable, type:

```
> c := 25;
```

Now the value 25 has been stored to the name `c`, and you can refer to this number by the name `c` in subsequent calculations.

Assume that `c` is 25° Celsius. If you want to convert it to Fahrenheit, enter:

```
> print(1.8*c + 32);
77
```

There are many functions available in the kernel and in various libraries. To compute the inverse sine, use the **arcsin** operator:

```
> print(arcsin(1));
1.5707963267949
```

The **root** function determines the n-th root of a value:

```
> print(root(2, 3));
1.2599210498949
```

3.4 Useful Statements

Instead of using **print**, you may also output results by entering an expression and completing it with a colon:

```
> root(2, 3):
1.2599210498949
```

The global variable **ans** always holds the result of the last statement you completed with a colon.

```
> ln(2*Pi):
1.8378770664093

> ans:
1.8378770664093
```

The console screen can be cleared by just entering the keyword **cls**:

```
> cls
```

The **restart** statement resets Agena to its initial state, i.e. clears all variables you defined in a session.

```
> restart
```

The **bye** statement quits a session - you can also press CTRL+C, alternatively.

```
> bye
```

If you would like to automatically run a procedure before restarting or quitting Agena, just assign this procedure to the name **environ.onexit**. See the description of the **bye** statement in Chapter 8 for more details.

If you prefer another Agena prompt instead of the predefined one, assign for example:

```
> _PROMPT := 'Agena$ '
Agena$ _
```

You may put this statement into the initialisation file in the Agena library or your home folder, if you do not want to change the prompt manually every time you start Agena. See Appendix A6 for further detail.

```
Agena$ restart;
```

3.5 Assignment and Unassignment

As we have already seen, to assign a number, say 1, to a variable called a, type:

```
> a := 1;
```

Variables can be deleted by assigning **null** or using the **clear** statement. The latter also immediately performs a garbage collection.

```
> a := null:
null

> clear a;

> a:
null
```

3.6 Arithmetic

Agena supports both real and complex arithmetic with the **+** (addition), **-** (subtraction), ***** (multiplication), **/** (division) and **^** (exponentiation) operators:

```
> 1+2:
3
```

Complex numbers can be entered using the **I** constant or the **!** operator:

```
> exp(1+2*I):
-1.1312043837568+2.4717266720048*I

> exp(1!2):
-1.1312043837568+2.4717266720048*I
```

3.7 Strings

A text can be put in single or double quotes:

```
> str := 'a string':
a string
```

Substrings are extracted by passing an index or index range:

```
> str[3], str[3 to 6]:
s      stri
```

Concatenation, search, and replacement:

```
> str := str & ' and another one, too':
a string and another one, too

> instr(str, 'another'):
14

> replace(str, 'and', '&'):
a string & another one, too
```

There are various other string operators and functions available.

3.8 Booleans

Agena features the **true**, **false**, and **fail** constants to represent Boolean values. **fail** may be used to indicate a failed computation. The operators <, >, =, <>, <=, and >= compare values and return either **true** or **false**. The operators **and, or, not, nand, nor, xor** and **xnor** combine Boolean values.

```
> 1 < 2:
true

> true or false:
true
```

You can also do arithmetic with numbers and Booleans where **true** depicts 1 and **false**, **fail** or **null** 0. Also, applying the unary minus operator to Booleans will convert them to either the numbers 0 or -1.

3.9 Tables

Tables are used to represent more complex data structures. Tables consist of zero, one or more key-value pairs: the key referencing to the position of the value in the table, and the value the data itself.

```
> tbl := [
>    1 ~ ['a', 7.71],
>    2 ~ ['b', 7.70],
>    3 ~ ['c', 7.59]
> ];
```

To get the subtable `['a', 7.71]` indexed with key 1, and the second value 7.71 in this first subtable, input:

```
> tbl[1]:
[a, 7.71]

> tbl[1, 2]:
7.71
```

The **insert** statement adds further values into a table.

```
> insert ['d', 8.01] into tbl
```

```
> tbl:
[[a, 7.71], [b, 7.7], [c, 7.59], [d, 8.01]]
```

Alternatively, values may be added by indexing:

```
> tbl[5] := ['e', 8.04];

> tbl:
[[a, 7.71], [b, 7.7], [c, 7.59], [d, 8.01], [e, 8.04]]
```

Of course, values can be replaced:

```
> tbl[3] := ['z', -5];

> tbl:
[[a, 7.71], [b, 7.7], [z, -5], [d, 8.01], [e, 8.04]]
```

Another form of a table is the dictionary, with indices that can be any kind of data - not only positive integers. Key-value pairs are entered with tildes.

```
> dic := ['donald' ~ 'duck', 'mickey' ~ 'mouse'];

> dic['donald']:
duck
```

3.10 Sets

Sets are collections of unique items: numbers, strings, and any other data except **null**. Any item is stored only once and in random order.

```
> s := {'donald', 'mickey', 'donald'}:
{donald, mickey}
```

If you want to check whether 'donald' is part of the set, just index it or use the **in** operator:

```
> s['donald']:
true

> s['daisy']:
false

> 'donald' in s:
true
```

The **insert** statement adds new values to a set, the **delete** statement deletes them.

```
> insert 'daisy' into s;

> delete 'donald' from s;

> s:
{daisy, mickey}
```

Three operators exist to conduct Cantor set operations: **minus**, **intersect**, and **union**.

3.11 Sequences

Sequences can hold any number of items except **null**. All elements are indexed with integers starting with number 1. Compared to tables, sequences are twice as fast when adding values to them. The **insert**, **delete**, indexing, and assignment statements as well as the operators described above can be applied to sequences, too.

```
> s := seq(1, 1, 'donald', true):
seq(1, 1, donald, true)

> s[2]:
1

> s[4] := {1, 2, 2};

> insert [1, 2, 2] into s;

> s:
seq(1, 1, donald, {1, 2}, [1, 2, 2])
```

3.12 Pairs

Pairs hold exactly two values of any type, including **null** and other pairs. Values can be retrieved by indexing them or using the **left** and **right** operators. Values may be exchanged by using assignments to indexed names.

```
> p := 10:11;

> left(p), right(p), p[1], p[2]:
10      11        10        11

> p[1] := -10;
```

3.13 Conditions

Conditions can be checked with the **if** statement. The **elif** and **else** clauses are optional. The closing **fi** is obligatory.

```
> if 1 < 2 then
>    print('valid')
> elif 1 = 2 then
>    print('invalid')
> else
>    print('invalid, too')
> fi;
valid
```

The **case** statement facilitates comparing values and executing corresponding statements.

There are two flavours: The first checks an expression for certain values.

```
> c := 'agena';
```

```
> case c
>    of 'agena' then
>        print('Agena!')
>    of 'lua' then
>        print('Lua!')
>    else
>        print('Another programming language !')
> esac;
Agena!
```

The second one works exactly like the **if** statement but may improve code readability.

```
> v := 1;

> case
>    of v > 0 then print(1)
>    of v = 0 then print(0)
>    else print(-1)
> esac;
1
```

3.14 Loops

A **for** loop iterates over one or more statements. It starts with an initial numeric value (**from** clause), and proceeds up to and including a given numeric value (**to** clause). The step size can also be given (**step** clause). The **od** keyword indicates the end of the loop body.

The **from** and **step** clauses are optional. If the **from** clause is omitted, the loop starts with the initial value 1. If the **step** clause is omitted, the step size is 1.

The current iteration value is stored to a control variable (i in this example) which can be used in the loop body.

```
> for i from 1 to 3 by 1 do
>    print(i, i^2, i^3)
> od;
1       1       1
2       4       8
3       9       27
```

A **while** loop first checks a condition and if this condition is **true** or any other value except **false**, **fail** or **null**, it iterates the loop body again and again as long as the condition remains **true**. The following statements calculate the largest Fibonacci number less than 1000.

```
> a := 0; b := 1;

> while b < 1000 do
>    c := b; b := a + b; a := c
> od;

> c:
987
```

A variation of while is the **do/as** loop which checks a condition at the end of the iteration. Thus the loop body will always be executed at least once.

```
> c := 0;

> do
>    inc c
> as c < 10;

> c:
10
```

All flavours of **for** loops can be combined with a **while** condition. As long as the **while** condition is satisfied, i.e. is **true**, the **for** loop iterates.

```
> for x to 10 while ln(x) <= 1 do
>    print(x, ln(x))
> od;
1    0
2    0.69314718055995
```

The **skip** statement starts another iteration of the loop immediately, thus skipping all of the following loop statements after the **skip** keyword for the current iteration.

The **break** statement quits execution of the loop and proceeds with the next statement right after the end of the loop. Thus the above loop could also be written as:

```
> for x to 10 do
>    if ln(x) > 1 then break fi;
>    print(x, ln(x))
> od;
1    0
2    0.69314718055995
```

which of course is equivalent to

```
> for x to 10 while ln(x) <= 1 do
>    print(x, ln(x))
> od
1       0
2       0.69314718055995
```

for loops can also be combined with a closing **as** or **until** condition. In this case, the loop body is always executed at least once. The loop is iterated as long as the **as** condition remains **true** or the **until** condition evaluates to **false**.

```
> for x to 10 do
>    print(x, ln(x))
> as ln(x) <= 1
1       0
2       0.69314718055995
3       1.0986122886681

> for x to 10 do
>    print(x, ln(x))
> until ln(x) > 1
```

```
1        0
2        0.69314718055995
3        1.0986122886681
```

3.15 Procedures

Procedures cluster a sequence of statements into abstract units which then can be run repeatedly.

Local variables are accessible to their procedure only and can be declared with the **local** statement.

The **return** statement passes the result of a computation.

```
> fact := proc(n) is
>    local result;
>    result := 1;
>    for i from 1 to n do
>        result := result * i
>    od;
>    return result
> end;

> fact(10):
3628800
```

A procedure can call itself.

If your procedure consists of exactly one expression, then you may use an abridged syntax if the procedure does not include statements such as **if**, **for**, **insert**, etc.

```
> deg := << (x) -> x * 180 / Pi >>;
```

To compute the value of the function at $\frac{\pi}{4}$, just input:

```
> deg(Pi/4):
45
```

Alternatively, you may use the **def** or the **define** statement, e.g. a function with two arguments can be defined as follows:

```
> define sum(x, y) -> x + y;

> sum(1, 2):
3
```

The **->** assignment token is optional. Alternatively, you can also use an = or := sign or the **is** keyword.

3.16 Comments

You should always document your code so that you and others will understand its purpose if reviewed later.

A single line comment starts with a single hash. Agena ignores all characters following the hash up to the end of the current line.

```
> # this is a single-line comment
> a := 1; # a contains a number
```

A multi-line comment, also called `long comment`, starts with the token sequence `#/` and ends with the closing `/#` token sequence[1].

```
> #/ this is a long comment,
>    split over two lines /#
```

Alternatively, C comments are supported, as well:

```
> /* this is a one-line comment */

> /* this is a long comment,
>    split over two lines */
```

3.17 Writing, Saving, and Running Programmes

While short statements can be entered directly at the Agena prompt, it is quite useful to write larger programmes in a text editor and save them to a text file so that they can be reused in future sessions.

Note that Agena comes with language scheme files for some common text editors. Look into the share/schemes subdirectory of your Agena installation.

Let us assume that a programme has been saved to a file called `myprog.agn` in the directory `/home/alex` in UNIX, or `c:\Users\alex` in OS/2, DOS or Windows. Then in UNIX, you can run it at the Agena prompt by typing:

```
> run '/home/alex/myprog.agn'
```

or in DOS-based systems:

```
> run 'c:/users/alex/myprog.agn'   or

> run 'c:\\users\\alex\\myprog.agn'
```

in DOS-based systems.

---

[1] Multi-line comments cannot begin in the very first line of a programme file. Use a single comment, i.e. `#`, instead.

If you both want to start an Agena session and also run a programme from a shell, then enter:

```
$ agena -i /home/alex/myprog.agn
```

in UNIX or

```
C:\>agena -i c:\users\alex\myprog.agn
```

in Windows. See Appendix A4 for further switches.

3.18 Using Packages

Many functions are included in additional packages which must at first be initialised so that the package functions can be used. Part II of this document indicates which packages are automatically initialised at Agena start-up and which packages have to be imported manually by the user.

For example, all statistics functions are included in the *stats* package which can be invoked with the **import** statement:

```
> import stats;

> stats.amean([1, 2, 3, 4]):
2.5
```

Shortcuts to the package functions can be defined by passing the **alias** option to the **import** statement.

```
> amean([1, 2, 3, 4]):
Error in stdin, at line 1:
   attempt to call global `amean` (a null value)

> import stats alias
Warning: iqr, sorted have been reassigned.

> amean([1, 2, 3, 4]):
2.5
```

If you want to define shortcuts to specific package functions only, pass their names right after the **alias** option:

```
> import stats alias amean, smm;
```

If you pass the **as** clause instead, it assigns an alias to a library name:

```
> import hashes as h;

> a := h.crc32('agena');
```

You may also have a look at the **readlib** and **initialise** functions described in Chapter 8.

Chapter **Four**

Data & Operations

4 Data & Operations

Agena features a set of data types and operations on them that are suited for both general and specialised needs. While providing all the general types inherited from Lua - numbers, strings, booleans, nulls, tables, and procedures - it also has four additional data types that allow very fast operations: sets, sequences, registers, pairs, and complex numbers.

Type	Description
number	any integral or rational number, plus **undefined** and **infinity**
string	any text
boolean	booleans (e.g. **true**, **false**, and **fail**)
null	a value representing the absence of a value
table	a multipurpose structure storing numbers, strings, booleans, tables, and any other data type
procedure	a predefined collection of one or more Agena statements
set	the classical Cantor set storing numbers, strings, booleans, and all other data types available
sequence	a dynamically-sized vector storing numbers, strings, booleans, and all other data types except **null** in sequential order
register	a fixed-size vector storing any value including **null** and featuring a top position pointer to prevent access to elements above it
pair	a pair of two values of any type
complex	a complex number consisting of a real and an imaginary number
userdata	part of system memory containing user-defined data; userdata objects can only be created by changing the ANSI C sources of the interpreter
lightuserdata	a value representing a C pointer; available only if you modify the ANSI C sources of the interpreter
thread	a non-preemptive multithread object (a coroutine)

Table 1: Available types

Tables, sets, sequences, registers, and pairs are also called *structures* in this manual.

You can determine the type of a value with the **type** operator which returns a string:

```
> type(0):
number

> type('a text'):
string
```

There is also a structure derived from both tables and sets: bags, see Chapter 10.8; also have a look on linked lists, see Chapter 10.7.

4.1 Names, Keywords, and Tokens

In Chapter 3, we have already assigned data - such as numbers and procedures - to names, also called `variables`. These names refer to the respective values and can be used conveniently as a reference to the actual data.

A name always begins with an upper-case or lower-case letter or an underscore, followed by one or more upper-case or lower-case letters, underscores, single quotes or numbers in any order.

Since Agena is a dynamically typed language, no declarations of variable names are needed.

Valid names	Invalid names
var	1var
_var	1_
var1	
_var1n	
_1	
ValueOne	
valueTwo	
Value'One	

Table 2: Examples for valid and invalid names

The following keywords are reserved and cannot be used as names:

```
abs alias and antilo2 antilog10 arccos arcsec arcsin arctan as assigned
atendof bea bottom break by bye case catch char cis clear cls conjugate
constant cos cosh cosxx create dec def define delete dict div do downto
duplicate elif else empty end entier enum esac esle even exchange exp
fail false feature fi filled first finite flip float for from global if
imag import in inc infinite infinity inrange insert int intdiv integral
intersect into invsqrt is keys last left ln lngamma local lower minus
mod mul muladd mulup nan nand nargs negate nonzero nor not notin numeric
od odd of onsuccess or pop proc procname qmdev qsumup real redo reg
relaunch reminisce restart return right rotate seq sign signum sin sinc
sinh size skip split sqrt square squareadd store subset sumup tan tanh
then to top trim true try type typeof unassigned undefined union until
upper when while with xnor xor xsubset yrt zero

anything boolean complex lightuserdata listing null number pair register
procedure sequence set string table thread userdata
integer nonnegative nonnegint posint positive
```

The following symbols denote other tokens:

```
+ - * ** / *% /% +% -% \ & && || ~ ~~ % %% ^ ^^ # = <> <= >= < > =
== ~= ~<> <<< >>> <<<< >>>> ( ) { } [ ] ; : :: :- -> @ @@ $ $$ , . .. ?
?- ` ++ -- +++ --- // \\ (/ \) | |- +:= -:= *:= /:= \:= %:= &:= &+ &- &*
&/ &\
```

4.2 Assignment

Values can be assigned to names in the following fashions:

> [**constant**] *name* := *value*
> [**constant**] *name₁*, · · ·, [**constant**] *nameₖ* := *value₁*, · · ·, *valueₖ*
> [**constant**] *name₁*, · · ·, [**constant**] *nameₖ* -> *value*

In the first form, one value is stored in one variable, whereas in the second form, called `multiple assignment statement`, $name_1$ is set to $value_1$, $name_2$ is assigned $value_2$, etc. In the third form, called the `short-cut multiple assignment statement`, a single value is set to each name to the left of the –> token.

First steps:

```
> a := 1;
```

```
> a:
1
```

An assignment statement can be finished with a colon to both conduct the assignment and print the right-hand side value at the console.

```
> a := 1:
1
```

```
> a := exp(a):
2.718281828459
```

Multiple assignments:

```
> a, b := 1, 2
```

```
> a:
1
```

```
> b:
2
```

If the left-hand side contains more names than the number of values on the right-hand side, then the excess names are set to **null**.

```
> c, d := 1
```

```
> c:
1
```

```
> d:
null
```

If the right-hand side of a multiple assignment contains extra values, they are simply ignored.

The multiple assignment statement can also be used to swap or shift values in names without using temporary variables.

```
> a, b := 1, 2;

> a, b := b, a:
2       1
```

A short-cut multiple assignment statement:

```
> x, y -> exp(1);

> x:
2.718281828459

> y:
2.718281828459
```

You can declare constants by putting the **constant** keyword in front of a variable name in an assignment. If you try to assign a new value to the constant later on in a session, the interpreter will issue an error:

```
> constant a := 1;

> a := 2;
Error at line 1: attempt to assign to constant `a` near `:=`
```

You can declare multiple constants at a time:

```
> constant b, constant c := 2, 3;

> b := 0;
Error at line 1: attempt to assign to constant `b` near `:=`

> c := 0;
Error at line 1: attempt to assign to constant `c` near `:=`
```

You can mix ordinary and constant declarations:

```
> a, constant b := 1, 2;
```

You should assign a value to a constant in one and the same declaration, otherwise you cannot use it:

```
> a, constant b := 1; # assign 1 to name `a`, and no value to constant `b`

> b := 0
Error at line 1: attempt to assign to constant `b`, near `:=`
```

You can switch off this feature completely with the following statement:

```
> environ.kernel(constants = false);
```

On the interactive level, if you define one and the same constant multiple times in a body, for example a **then** or **do** body, Agena will just print a one-time warning message but will change this constant. When executing a script file, however, Agena will exit with a proper error message. This is due to the way the parser evaluates bodies on the command-line. Also, in closures (see Chapter 6.22) constants cannot be recognised, so if you try to change them, no error will be issued.

4.3 Enumeration

Enumeration with step size 1 is supported with the **enum** statement:

$$\textbf{enum } name_1 \text{ [, } name_2, \cdots \text{]}$$
$$\textbf{enum } name_1 \text{ [, } name_2, \cdots \text{] } \textbf{from } value$$

All these values are constants, you cannot change them later on.

In the first form, $name_1$, $name_2$, etc. are enumerated starting with the numeric value 1.

```
> enum ONE, TWO;

> ONE:
1

> TWO:
2
```

In the second form, enumeration starts with the numeric value passed right after the **from** keyword.

```
> enum THREE, FOUR from 3

> THREE:
3

> FOUR:
4
```

4.4 Deletion and the null Constant

You may delete the contents of one or more variables with one of the following methods: Either use the **clear** command,

$$\textbf{clear } name_1 \text{ [, } name_2, \cdots, name_k \text{]}$$

```
> a := 1;

> clear a;

> a:
null
```

which also performs a garbage collection useful if large structures shall be immediately removed from memory, or set the variable to be deleted to **null**:

```
> b := 1;

> b := null:
null
```

The **null** value represents the absence of a value. All names that are unassigned evaluate to **null**. Assigning names to **null** quickly clears their values, but does not garbage collect them immediately.

The **null** constant has its own type: '**null**'.

```
> type(null):
null
```

If you want to test whether a value is of type 'null', contrary to all other types, you have to put the type name in brackets:

```
> type(null) = 'null':
true
```

In all cases - whether using the **clear** statement or assigning to **null** - the memory freed is not given back to the operating system but can be used by Agena for values yet to be created.

There are two operators that quickly check whether a value is assigned or not: **assigned** and **unassigned**.

```
> assigned(v):
false

> unassigned(v):
true
```

4.5 Precedence

Operator precedence in Agena follows the table below, from lower to higher priority:

```
or xor nor xnor
and nand
<  >  <= >= = == ~= ~<> <> :: :- |
in notin subset xsubset union minus intersect atendof |-
& : @ $ $$
+ - || ^^ split &+ &- inc dec
* / % symmod roll \ && *% /% %% +% -% %% <<< >>> <<<< >>>> &* &/ &\
  squareadd mul div intdiv mod
```
not - (unary minus) `+++ ---`
```
^ **
```
`!` and all self-defined binary operators and unary operators including `~~`

As usual, you can use parentheses to change the precedence of an expression. The concatenation (`&`), exponentiation (`^`, `**`), pair (`:`), mapping (`@`), and selection (`$`) operators are right associative, e.g. x^y^z = x^(y^z). All other binary operators are left associative.

```
> 1+3*4:
13

> (1+3)*4:
16
```

4.6 Arithmetic

4.6.1 Numbers

In the `real` domain, Agena internally only knows floating point numbers which can represent integral or rational numeric values. All numbers are of type **number**.

An integral value consists of one or more numbers, with an optional sign in front of it.

- 1
- -20
- 0
- +4

A rational value consists of one or more numbers, an obligatory decimal point at any position and an optional sign in front of it:

- -1.12
- 0.1
- .1

Negative integral or rational values must always be entered with a minus sign, but positive numbers do not need to have a preceding plus sign.

You may optionally include one or more single quotes or underscores *within* a number to group digits:

```
> 10'000'000:
10000000
```

You can alternatively enter numbers in scientific notation using the `e` symbol.

```
> 1e4:
10000
```

```
> -1e-4:
-0.0001
```

If a number ends in the letter `K`, `M`, `G`, `T` or `D`, then the number is multiplied by 1,024, 1,048,576 (= $1,024^2$), 1,073,741,824 (= $1,024^3$), 1,099,511,627,776 (= $1,024^4$), or 12, respectively. If a number ends in the letter `k`, `m`, `g`, `t` or `d`, then the number is multiplied by 1,000, 1,000,000, 1,000,000,000, 1,000,000,000,000, or 12 respectively. If a number is appended by `p`, it is converted to percentage.

```
> 2k:
2000
```

```
> 1M:
1048576
```

```
> 12D:
144
```

Besides decimal numbers, Agena supports binary, octal, and hexadecimal numbers which may include `thousands` separators. They are represented by the first two letters `0b` or `0B`, `0o` or `0O`, `0x` or `0X`, respectively:

System	Syntax	Examples (to decimal)
binary	`0b<binary number>` or `0B<binary number>`	`0b10` = 2
octal	`0o<octal number>` or `0O<octal number>`	`0o10` = 8
hexadecimal (integer)	`0x<hexadecimal number>` or `0X<hexadecimal number>`	`0xa` = 10
hexadecimal (float)	`0x<int>.<frac>` or `0X<int>.<frac>E` or `0x<int>p<int>` or `0x<int>.<frac>P<int>`	0x0.1 = 0.0625 0x0.1E = 0.0625 0xa23p-4 = 162.1875 0X1.921FB54442D18P+1 = 3.1415926535898

If you use only real numbers in your programmes, then Agena will calculate only in the real domain. If you use at least one complex value (see Chapter 4.6.5), then Agena will calculate in the complex domain.

Since Agena internally stores numbers in double or complex double precision, you will sometimes encounter round-off errors. For example, some values such as $\sqrt{2}$ or $\frac{1}{3}$ cannot be accurately represented on a machine.

The **mapm** package can be used in such situations as it provides arbitrary precision arithmetic. See Chapter 11.3 for more information.

Agena knows two representation for zero: 0 and -0, where -0 means something like zero but `approached from` $-\infty$. In relations, 0 and -0 are always the same, e.g. 0 = -0 $\Rightarrow$ **true**, and 0 < -0 $\Rightarrow$ **false**. In arithmetic, for example -1 * -0 $\Rightarrow$ -0. To test for -0, use **math.isminuszero**.

4.6.2 Arithmetic Operations

Agena has the following arithmetical operators:

Operator	Operation	Details / Example
+	Addition	`1 + 2 » 3`
−	Subtraction	`3 – 2 » 1`
*	Multiplication	`2 * 3 » 6`
/	Division	`4 / 2 » 2`
^	Exponentiation with rational power	`2 ^ 3 » 8`
**	Exponentiation with integer power	`2 ** 3 » 8`
%	Modulus	`5 % 2 » 1`
\	Integer division	`5 \ 2 » 2`
*%	Percents, percentage	`100 *% 2 » 2`
/%	Percents, ratio	`100 /% 2 » 5k`
+%	Percents, add-on (premium)	`100 +% 2 » 102`
−%	Percents, discount	`100 –% 2 » 98`
@	Conditional multiplication a @ b, returning a if b = 0, and a*b otherwise	`2 @ 0 » 2` `2 @ 3 » 6`

Table 3: Arithmetic operators

The modulus operator is defined as a % b = a - entier(a/b)*b, the integer division as a \ b = sign(a) * sign(b) * entier(abs(a/b)).

Agena has a lot of mathematical functions both built into the kernel and also available in the **math**, **stats**, **linalg**, and **calc** libraries. Table 4 lists some of the most common.

The mathematical procedures that reside in packages must always be entered by passing the name of the package followed by a dot and the name of the procedure. Use the **import** statement to activate the package before using these functions, e.g. to initialise the statistics package called *stats*, type:

```
> import stats;
```

Unary operators[2] like **ln**, **exp**, etc. can be entered with or without simple brackets.

Procedure	Operation	Library	Example and result
sin(x)	Sine (x in radians)	Kernel	`sin 0 » 0`
cos(x)	Cosine (x in radians)	Kernel	`cos(0) » 1`
tan(x)	Tangent (x in radians)	Kernel	`tan(1) » 1.557407..`
arcsin(x)	Inverse sine (x in radians)	Kernel	`arcsin(0) » 0`
arccos(x)	Arc cosine (x in radians)	Kernel	`arccos(0) » 1.570796..`
arctan(x)	Arc tangent (x in radians)	Kernel	`arctan(Pi) » 1.262627..`
sinh(x)	Hyperbolic sine	Kernel	`sinh(0) » 0`
cosh(x)	Hyperbolic cosine	Kernel	`cosh(0) » 1`
tanh(x)	Hyperbolic tangent	Kernel	`tanh(0) » 0`
abs(x)	Absolute value of x	Kernel	`abs(-1) » 1`
entier(x)	Rounds x downwards to the nearest integer	Kernel	`entier(2.9) » 2` `entier(-2.9) » -3`
even(x)	Checks whether x is even	Kernel	`even(2) » true`
exp(x)	Exponentiation e^x	Kernel	`exp(0) » 1`
lngamma(x)	$\ln \Gamma\, x$	Kernel	`exp(lngamma(3+1)) » 6`
int(x)	Rounds x to the nearest integer towards zero	Kernel	`int(2.9) » 2` `int(-2.9) » -2`
ln(x)	Natural logarithm	Kernel	`ln(1) » 0`
log(x, b)	Logarithm of x to the base b	Kernel	`log(8, 2) » 3`
round(x, d)	Rounds the real value x to the d-th digit	Base	`round(` `  sqrt(2), 2) » 1.41`
sign(x)	Sign of x	Kernel	`sign(-1) » -1`
sqrt(x)	Square root of x	Kernel	`sqrt(2) » 1.414213..`
sumup([···])	Sum	Kernel	`sumup([1, 2, 3]) » 6`
mean([···])	Arithmetic mean	stats	`stats.mean([1, 2, 3]) » 2`
median([···])	Median	stats	`stats.median(` `  [1, 2, 3, 4]) » 2.5`

Table 4: Common mathematical functions

In addition, Agena can conduct bitwise operations on numbers.

Operator	Operation	Details / Example
&&	Bitwise `and`	`7 && 2 » 2`
||	Bitwise `or`	`1 || 2 » 3`
^^	Bitwise `exclusive-or`	`7 ^^ 2 » 5`
~~	Bitwise complement (bitwise `not`)	`~~7 » -8`
<<<, >>>	Bitwise shift	<<< conducts a left-shift (multiplication with 2), >>> a right-shift (division by 2).
<<<<, >>>>	Bitwise rotation	<<<< and >>>> rotate bits left- and rightwards.
nand	bitwise complement	Equivalent to ~~(a && b).

---

[2] See Appendix A1 for a list of all unary operators.

Operator	Operation	Details / Example
	`and`	
nor	bitwise complement `or`	Equivalent to ~~(a || b)
xnor	bitwise complement exclusive-`or`	Equivalent to ~~(a ^ ^ b)
getbit getbits	returns stored bit(s)	getbit(3, 1), getbits(3)
setbit setbits	sets bit(s)	setbit(3, 1) » 1, setbits(8, reg(1, 0, 0)) » 12

Table 5: Bitwise operators and functions

By default, the operators internally calculate with unsigned integers. You can change this behaviour to signed integers with **environ.kernel**:

```
> environ.kernel(signedbits = true);
```

The default is restored as follows:

```
> environ.kernel(signedbits = false);
```

Note that in order to return useful results ~~, **nand**, **nor** and **xnor** should be used in signed mode only, regardless of the **environ.kernel/signedbits** setting.

4.6.3 Increment, Decrement, Multiplication, Division

Instead of incrementing or decrementing a value, say

```
> a := 1;
```

by entering a statement like

```
> a := a + 1:
2
```

you can use the **inc** and **dec** commands[3] which are also around 10% faster:

> **inc** *name* [, *value*];
> **dec** *name* [, *value*];

If *value* is omitted, *name* is increased or decreased by 1.

```
> inc a;

> a:
3
```

---

[3] Finishing an **inc** or **dec** statement with a colon instead of a semicolon is refused.

```
> dec a;

> a:
2

> inc a, 2;

> a:
4
> dec a, 3;

> a:
1
```

Likewise, the **mul** and **div** statements multiply or divide their argument by a scalar, **mod** takes the modulus, and **intdiv** conducts an integer division, their defaults also being 1. **negate** flips a Boolean; with numbers, it converts 0 to 1, and non-zero to 0.

It is advised that all **inc**, **dec**, **mul**, **div**, **intdiv** and **mod** statements are terminated by a semicolon unless the next token in the code is a keyword, so that the parser can discern them from the corresponding operators, see Chapter 4.6.8.

Alternatively, you may use mutate operators to express compound assignment:

Operator	Operation	Equivalent
+:=	addition	**inc** statement
-:=	subtraction	**dec** statement
*:=	multiplication	**mul** statement
/:=	division	**div** statement
\:=	integer division	**intdiv** statement
%:=	modulus	**mod** statement
&:=	string concatenation	n/a
@:=	conditional multiplication	n/a

```
> a +:= 3;  # equals to `a := a + 3` or `inc a, 3`

> a:
4
```

The suffix ++ and -- operators return the current value of a variable and *subsequently* increase or decrease the variable by one. Likewise, the prefix ++ and -- operators first increase or decrease a variable by one and then return the updates value. The operators work on indexed names, as well.

```
> c := 0;

> a := c++;      # used as an expression

> print(a, c);  # returns 0, 1

> c++;          # used as a statement

> print(c)
2
```

4.6.4 Mathematical Constants

Agena features arithmetic constants mentioned in Appendix A9. All mathematical constants are protected and cannot be changed.

All mathematical functions and operators return the constant **undefined** instead of issuing an error if they are not defined at a given point:

```
> ln(0):
undefined
```

With values of type number, the **finite** function can determine whether a value is neither ±**infinity** nor **undefined**.

```
> finite(fact(1000)), finite(sqrt(-1)):
false    false
```

The **float** function checks whether a value is a float and not an integer.

```
> float(1):
false

> float(1.1):
true
```

4.6.5 Complex Math

Complex numbers can be defined in two ways: by using the `!` constructor or the imaginary unit represented by the capital letter `I`. Most of Agena's mathematical operators and functions know how to handle complex numbers and will always return a result that is in the complex domain. Complex values are of type **complex**.

```
> a := 1!1;

> b := 2+3*I;

> a+b:
3+4*I

> a*b:
-1+5*I
```

The following operators work on rational numbers as well as complex values: `+`, `-`, `*`, `/`, `^`, `**`, `=`, `<>`, `abs`, `arccos`, `arcsec`, `arcsin`, `arctan`, `conjugate`, `cos`, `cosh`, `entier`, `exp`, `flip`, `invsqrt`, `lngamma`, `ln`, `log`, `sign`, `sin`, `sinh`, `sqrt`, `tan`, `tanh`, and unary minus. With these operators, you can also mix numbers and complex numbers in expressions. You will find that most mathematical functions are also applicable to complex values.

```
> c := ln(-1+I) + ln(0.5):
-0.34657359027997+2.3561944901923*I
```

The real and imaginary parts of a complex value can be extracted with the **real** and **imag** operators.

```
> real(c), imag(c):
-0.34657359027997       2.3561944901923
```

Three further functions may also be of interest: **abs** returns the absolute value of a complex number, **argument** returns its phase angle in radians, and **conjugate** computes the complex conjugate.

Note that the `!` operator has the same precedence as unary operators like `-`, `sin`, `cos`, etc. This means that `-1!2 = -1+2*I`, but also that `sin 1!2 = (sin 1)!2`. Thus, it is advised that you use brackets when applying unary operators on complex values.

The setting `environ.kernel(zeroedcomplex = true)` makes Agena *print* complex values that are close to zero as just `0` in the output region of the console. Internally, however, complex values are not rounded by this or any other setting.

4.6.6 Comparing Values

Relational operators can compare both numeric and complex values. Whereas all relational operators work on numbers, complex numbers can only be compared for equality or inequality.

Operator	Description	Complex values supported
<	less than	no
>	greater than	no
<=	less than or equals	no
>=	greater than or equals	no
=	equals	yes
<>	not equals	yes
in	in range	no

```
> 1 < 2:
true

> 1 = 1:
true

> 1 <> 1:
false
```

The result **true** indicates that a comparison is valid, and **false** indicates that it is invalid. See Chapter 4.8 for more information.

Most computer architectures cannot accurately store number values unless they can be expressed as halves, quarters, eighths, and so on. For example, 0.5 is represented accurately, but 0.1 or 0.2 are not.

Since Agena is not a computer algebra system, you will sometimes encounter round-off errors in computations with numbers and complex numbers:

```
> 0.2 + 0.2 + 0.2 = 0.6:
false
```

In such cases, the ~= operator or the **approx** function might be of some help since they compare values approximately.

```
> 0.2 + 0.2 + 0.2 ~= 0.6:
true

> 0.2!0.2 + 0.2!0.2 + 0.2!0.2 = 0.6!0.6:
false

> approx(0.2!0.2 + 0.2!0.2 + 0.2!0.2, 0.6!0.6):
true
```

To determine whether a number is part of a closed interval, use the **in** or **inrange** operators:

```
> 2 in 0 : 10:
true
```

You can use the **+++** and **---** operators to define open borders:

```
> inrange(1, +++1, ---10):
false
```

The unary **zero** operator checks whether a number or complex number is 0 or $0+I*0$; **nonzero** checks whether it is non-zero. The two operators are around 10 % faster than the binary = and <> operators.

4.6.7 Range of Values

The following ranges apply to Agena numbers and complex numbers:

Characteristic	Value
smallest representable number	$-1.797\ 693\ 134\ 862\ 315 \times 10^{308}$
largest representable number	$+1.797\ 693\ 134\ 862\ 315 \times 10^{308}$
largest positive integer without loss of precision	$9.007199254741 \times 10^{15}$
smallest subnormal (negative) positive number	(-)4.9406564584124654e-324
largest subnormal (negative) positive number	(-)2.2250738585072009e-308

4.6.8 Adapting Basic Arithmetic Operators

There are six arithmetic binary operators that detect potential numeric overflow, underflow and division by zero and allow the user to invoke proper self-written functions that handle them: **inc** for addition, **dec** for subtraction, **mul** for multiplication, **div** for division and **intdiv** for integer division, plus **mod** for modulus.

These operators after checking possible exceptions call user-defined handlers that take the two operands plus the information on the kind of exception:

- addition: **inc** calls **math.add**,
- subtraction: **dec** calls **math.subtract**,
- multiplication: **mul** calls **math.multiply**,
- division: **div** calls **math.divide**,
- integer division: **intdiv** calls **math.intdivide** and
- modulus: **mod** calls **math.modulo** (not **math.modulus** !).

Examples:

Division: the handler might look like this - **math.intdivide** and **math.modulo** may look similar, since the values for parameter kind are the same:

```
> math.divide := proc(n, d, kind) is
>    case kind
>           # kind 0b0000 means no exception
>       of 0b0000 then return n/d
>           # kind 0b0001 means denominator is zero
>       of 0b0001 then error('division by zero')
>           # kind 0b0010 means very large value to be divided by value
>           # close to 0
>       of 0b0010 then error('underflow')
>           # kind 0b1000 indicates both operands are close to 0
>       of 0b1000 then return n/d
>    esac
> end;

> 1 div 0:
division by zero
```

A multiplication handler:

```
> math.multiply := proc(a, b, kind) is
>    case kind
>       of 0b0000 then return a*b  # kind 0b000 indicates no exception
>           # kind 0b0010: very large value to be multiplied by value close
>           # to zero
>       of 0b0010 then error('underflow')
>           # kind 0b0100: very large value to be multiplied by
>           # very large value
>       of 0b0100 then error('overflow')
>           # kind 0b1000: both operands are close to zero
>       of 0b1000 then return a*b
>    esac
> end;
```

```
> 1e308 mul 1e308:
overflow
```

Addition - and subtraction if this should make any sense, the possible values for `kind` are the same - could be handled like this (for subtraction redefine **math.subtract**):

```
> math.add := proc(a, b, kind) is
>    case kind
>       of 0b0000 then return a + b  # no exception
>          # very large value to be added to (subtracted) value close to 0:
>       of 0b0010 then return a + b
>          # very large values to be added (or subtracted):
>       of 0b0100 then error('overflow')
>          # both operands are close to zero:
>       of 0b1000 then return a + b
>    esac
> end;
```

Agena is shipped with six default functions **math.add**, **math.subtract**, **math.multiply**, **math.divide**, **math.intdivide** and **math.modulo** that just conduct the requested operation and return the result, without issuing any error. You may overwrite them with alternatives of your choice.

The threshold that defines whether a value is `close to zero` can be set with **environ.kernel/closetozero**, which by default is **DoubleEps**, e.g.:

```
> environ.kernel(closetozero = 1e-20);
```

The type of numerical exception that occurred the last time one the six operators has been invoked can also be queried by calling **environ.arithstate** which returns the type of exception as a bit field, see all the **case/of** clauses above:

```
> 1e308 inc 1e308:
overflow

> environ.arithstate():
1
```

The description of **environ.arithstate** in Chapter 14.2 includes a complete list of all the numeric exceptions the six binary operators might encounter.

4.7 Strings

4.7.1 Representation

Any text can be represented by including it in single or double quotes:

```
> 'This is a string':
This is a string
```

Of course, strings - like numbers - can be assigned to variables.

```
> str := "I am a string.";

> str:
I am a string.
```

Strings - regardless whether included in single or double quotes - are all of type **string**,

```
> type(str):
string
```

and can be of almost unlimited length. Strings can be concatenated, characters or sequences of characters can be replaced by other ones, and there are various other functions to work on strings.

Multiline-strings can be entered by just pressing the RETURN key at the end of each line:

```
> str := 'Two
lines';
```

which prints as

```
> str:
Two
lines
```

A string may contain no text at all - called an *empty string* -, represented by two consecutive single quotes with no spaces or characters in between:

```
> '':
```

4.7.2 Substrings

You may obtain a specific character by passing its position in square brackets right after the string name. If you use a negative index n, then the |n|-th character from the right end of the string will be returned.

```
> str := 'I am a string.';
```

```
> str[1];
I
```

In general, parts of a string consisting of one or more consecutive characters can be obtained with the notation:

*string***[** *start* [**to** *end*] **]**

You must at least pass the start position of the substring. If only *start* is given then the single character at position *start* will be returned. If *end* is given too, then the substring starting at position *start* up to and including position *end* will be returned.

```
> str := 'string'

> str[3]:
r

> str[3 to 5]:
rin

> str[3 to 3]:
r
```

You may also pass negative values for *start* and/or *end*. In these cases, the positions are determined with respect to the right end of the string.

```
> str[3 to -1]:
ring

> str[3 to -2]:
rin

> str[-3 to -2]:
in

> str[-3]:
i
```

4.7.3 Escape Sequences

In Agena, a text can include any escape sequences[4] known from ANSI C, e.g.:

- `\n`: inserts a new line,
- `\t`: inserts a tabulator
- `\b`: puts the cursor one position to the left but does not delete any characters.

```
> 'I am a string.\nMe too.':
I am a string.
Me too.

> 'These are numbers: 1\t2\t3':
These are numbers: 1    2       3
```

---

[4] See also Appendix A7.

```
> 'Example with backspaces:\b but without the colon.':
Example with backspaces but without the colon.
```

If you want to put a single or double quote into a string, put a backslash right in front of it:

```
> 'A quote: \'':
A quote: '

> "A quote: \"":
A quote: "
```

However, if a string is delimited by single quotes and you want to include a double quote (or vice versa), a backslash is not obligatory, e.g. `"'agena'"` is a valid string.

Likewise, a backslash is represented by typing it twice.

4.7.4 Concatenation

Two or more strings can be concatenated with the `&` operator:

```
> 'First string, ' & 'second string, ' & 'third string':
First string, second string, third string
```

Numbers (but not complex ones) are supported, as well, so you do not need to convert them with the **tostring** function before applying `&`:

```
> 1 & ' duck':
1 duck
```

Furthermore, the compound `&:=` concatenation operator appends a string to the contents of a string variable:

```
> a := 'In';

> a &:= 'Sight';

> a:
InSight
```

4.7.5 String Operators and Functions

Agena has basic operators useful for text processing:

Operator	Return	Function
s **in** t	number or **null**	Checks whether a substring s is included in string t. If true, the position of the first occurrence of s in t will be returned; otherwise **null** will be returned.
s **atendof** t	number or **null**	Checks whether a string t ends in a substring s. If true, the position of the position of s in t will be returned; otherwise **null** will be returned.
replace(s, p, r)	string	Replaces all patterns p in string s with substring r. If p is not in s, then s will be returned unchanged. p might also be the position (a positive integer) of the character to be replaced.
s **split** d	sequence of strings	Splits a string into its words with d as the delimiting character(s). The items are returned as a sequence of strings.
size(s)	number	Returns the length of string s. If s is the empty string, 0 will be returned.
abs(s)	number	Returns the numeric ASCII code of character s.
char(n)	string	Returns the character corresponding to the given numeric ASCII code n.
lower(s)	string	Converts a string to lowercase. Western European diacritics are recognised.
upper(s)	string	Converts a string to uppercase. Western European diacritics are recognised.
tonumber(s)	number or complex value	Converts a string into a number or complex number.
tostring(n)	string	Converts a number to one string. If a complex value is passed, the real and imaginary parts are returned separately as two strings.
trim(s)	string	Deletes leading and trailing spaces as well as excess embedded spaces.

Table 7: String operators

Some examples:

```
> str := 'a string';
```

The character s is at the third position:

```
> 's' in str:
3
```

Let us split a string into its components that are separated by white spaces:

```
> str split ' ':
seq(a, string)
```

`str` is eight characters long:

```
> size(str):
8
```

The ASCII code of the first character in `str`, `a`, is:

```
> abs(str[1]):
97
```

translated back to

```
> char(ans):
a
```

Put all characters in `str` to uppercase:

```
> upper(str):
A STRING
```

And now the reverse:

```
> lower(ans):
a string
```

The following functions can be used to find and replace characters in a string:

Function	Functionality	Example
in	Returns the first position of a substring (left operand) in a string (right operand); if the substring cannot be found, **it** returns **null**.	`'tr' in 'string' » 2`

instr	Looks for the first match of a pattern (second argument) in a string (first argument). If it finds a match, then **instr** returns its position; otherwise, it returns **null**. An optional numerical argument specifies where to start the search. The function supports pattern matching, almost similar to regular expressions. The function is more than twice as fast as **strings.find**. If **true** is given as a fourth argument, pattern matching is switched off to speed up the search.
	
	If the option 'reverse' is given, then starting from the right end and always running to its left beginning, the function looks for the
		`instr(`
		`    'agena',`
		`    '[aA]g',`
		`    1) » 1`
		
		`instr('agena', 'a',`
		`    'reverse') » 5`

Function	Functionality	Example
	first match of the substring and returns the position where the pattern starts with respect to its left beginning. When searching from right to left, pattern matching is not supported.	
atendof	Checks whether a string (right operand) ends in a substring (left operand). If true, the position will be returned; otherwise **null** will be returned.	```'ing' atendof 'raining' » 5```
strings.find	Returns the first match of a substring (second argument) in a string (first argument) and returns the positions where the pattern starts *and ends*. An optional third argument specifies the position where to start the search. If it does not find a pattern, the function returns **null**. The function supports pattern matching facilities described in Chapter 9.1.3. See also: **strings.mfind**. which returns all occurrences.	```strings.find('string', 'tr') » 2, 3 strings.find('string', 'tr', 3) » null strings.find('string', 't.') » 2, 3```
replace	In a string (first argument) replaces all occurrences of a substring (second argument) with another one (third argument) and returns a new string. Pattern matching facilities are not supported. A sequence of replacement pairs can be passed to the function, too. See also **strings.gsub**.	```replace(str, 'string', 'text') » text replace('string', seq('s':'S', 't':'T')) » STring```

Table 8: Search and replace functions and operators

For more information on these functions, check Chapter 9.1. See also the descriptions of **strings.match** and **strings.gmatch**.

The replace function can be used to find and replace characters in a string.

4.7.6 Comparing Strings

Like numbers, single or multiple character strings can be compared with the familiar relational operators based on their sorting order which is determined by your current locale.

```
> 'a' < 'b':
```

```
true
> 'aa' > 'bb':
false
```

If the sizes of two strings differ, the missing character is considered less than an existing character.

```
> 'ba' > 'b':
true
```

4.7.7 Patterns and Captures

Sometimes it my not suffice to just look for a fixed pattern, e.g. a simple substring, in a string. You may want to search for a pattern of different kinds of characters - e.g. both numbers and letters, or either letters or numbers, or a subset of them -, or of variable number of characters, or both of them.

Agena provides both character classes and modifiers to accomplish this. While common Regular Expressions are not supported, Agena offers quite similar facilities, all taken from Lua.

For performance reasons, you may use the following rule of thumb[5]:

- If you would like to determine the start position of the very first match of a *fixed* pattern only, use the **in** operator, for **in** is the fastest.
- If you want to look as fast as possible only for the start position of the very first match of a `variable` pattern, using character classes and/or modifiers, or would like to give the position where to start the quickest search, use **instr**.
- If both the start and end position is needed, prefer **strings.find**. The **instr** function can also return the start and end position, with or without variable patterns, but may be slower than **strings.find** in most situations.

Character classes represent certain sets of tokens, e.g. the class `%d` represents one digit, and `%a` represents one upper-case or lower-case letter. Assume we would like to determine the position of the hour `00:00:00` in the following date/time string:

```
> date := '23.05.1949 00:00:00'
```

We could use the **instr** function to determine the start position of the hour,

```
> instr(date, '%d%d:%d%d:%d%d'):
12
```

or **strings.find** to get the start and end position of it.

```
> strings.find(date, '%d%d:%d%d:%d%d'):
12       19
```

---

[5] Different kinds of pattern matching facilities have been introduced in Agena deliberately, for the kind of search can significantly influence performance when processing a large number of strings. If you want to parse a large number of files and know where to look, **io.skiplines** may boost performance on slow drives, as well.

strings.match extracts the hour.

```
> strings.match(date, '%d%d:%d%d:%d%d'):
00:00:00
```

For a complete list of all supported classes, please have a look at the end of this chapter or Chapter 9.1.3.

Character sets denote user-defined classes comprising any character class and/or single tokens, put in square brackets. For example, `[01]` may represent a binary, and `[%l -]` any lower-case letter, white space or hyphen. A range of characters is represented by a hyphen, thus `[A-Ca-c]` represents one of the first three upper and lower case letters in the alphabet.

```
> instr('binary: 10', '[01]'):
9
```

A caret in front of a class indicates that a string should begin with this class, and a dollar trailing a class denotes that it should end in the given class.

```
> instr('1 is a number', '^[%l ]'):
null

> instr('1 is a number', '%l$'):
13
```

Patterns also support modifiers for repetition or optional parts. The plus sign indicates one or more repetitions of a class, the asterisk denotes zero or more repetitions, and the question mark zero or one occurrence.

```
> date := '23.05.1949 00:00:00'

> strings.find(date, '%d+.%d+.%d+'):  # find the date 23.05.1949
1       10

> date := '23.05. 00:00:00'

> strings.find(date, '%d+.%d+.%d*'):  # find 23.05., optionally the year
1        6
```

The single dot represents any occurrence of any character in a string, regardless whether the character is a cipher, a letter or special character. If you would like to search for one of the special characters `*`, `+`, `-`, `?`, `.`, `[`, `]`, etc. in a string, just escape it with the percentage sign.

```
> instr(date, '%.'):  # find the first dot in the date string
3
```

instr and **strings.find** also allow to switch off pattern matching by passing **true** as the last argument:

```
> instr(date, '.', true):
3
```

If a pattern is put in parentheses, one or more portions of a string matching this pattern are extracted from a string, to be optionally assigned to names. This feature is also called a capture. Two examples:

```
> strings.match('<id>1234</id>', '<id>(.*)</id>'):
1234

> date := 'May 23, 1949 12:15:00';

> strings.find(date, '(%w+) (%d+), ?(%d+)'):
1       12      May     23      1949

> year, day, month := strings.match(date, '(%w+) (%d+), ?(%d+)'):
May     23      1949

> year, month, day:
May     1949    23
```

Another useful function is **strings.gmatch** which returns a function that iterates over all occurrences of a pattern in a string:

```
> f := strings.gmatch('1 10', '(%d+)'):
procedure(008E1278)

> f():
1

> f():
10
```

You may also use the wrapper function **strings.gmatches** which returns a sequence of all the substrings matching a given pattern.

```
> strings.gmatches('1 10', '(%d+)'):
seq(1, 10)
```

There is a small difference between the * and – modifiers for matching zero or more occurrences which may influence execution time significantly: while * looks for the longest match, – does for the shortest:

```
> strings.match('<p>a</p><p>2</p>', '<p>(.-)</p>'):  # – shortest
a

> strings.match('<p>a</p><p>b</p>', '<p>(.*)</p>'):  # * longest
a</p><p>b
```

With captures, and with captures only, **strings.find** not only returns the start and end position of the match, but also the match itself as a third return.

```
> strings.find('<p>a</p><p>b</p>', '<p>(.-)</p>'):
1       8       a
```

To check whether one of the characters is in a given set, use square brackets. In the next example, we check whether the first character in a pattern is either '1', '2' or '3', and the rest of the pattern is 'abc'.

```
> strings.match('2abc', '[123]abc'):
2abc
```

The pattern in the above example, e.g. its second argument, in general matches a substring in a string. If you would like to make sure that a pattern matches an entire string, put a caret in front of the pattern and a dollar sign at its end:

```
> strings.match('2abc', '^[123]abc$'):
2abc
```

Thus, since the string to be searched is longer,

```
> strings.match('y2abcy', '^[123]abc$'):
```

returns:

```
null
```

To recognise one or more ligatures and umlauts, along with one or more Latin letters, also just use square brackets and combine them with a modifier, here `%a`:

```
> strings.match('Eckernförde, Schleswig-Holstein', '([äöüßÄÖÜ%a]*)'):
Eckernförde
```

Retrieve a value either residing in a conventional XML tag or its worst-case (though here invalid) SOAP variant:

```
> pattern := '<.*Data.*>(%a+)</.*Data>';

> str := strings.match(
>     '<soap:Data attr=\'foo\'>value</soap:Data>',
>     pattern);

> str:
value

> str := strings.match('<Data>value</Data>', pattern);

> str:
value
```

Summary[6] of character classes and pattern modifiers:

Classes		
	.	any character
	%a	letters a to z or A to Z
	%A	anything not matching the letters a to z or A to Z
	%c	control characters
	%C	anything not matching control characters
	%d	digits 0 to 9
	%D	anything not matching digits 0 to 9
	%i	an integer, consisting of one or more characters, optionally including a sign
	%k	upper and lower-case consonants (y is considered a vowel)
	%K	anything not matching upper and lower-case consonants
	%l	lower-case letters
	%L	anything not matching lower-case letters
	%n %N	a number, consisting of one or more characters, optionally including a preceding sign, a fractional part and a scientific E-notation suffix; a number may also just start with a sign and a fractional part. Optional decimal separator is always the dot with %n, and a comma with %N.
	%o	letters a to z or A to Z including diacritics and ligatures (provided Latin-1 codepage is active)
	%O	anything not matching the letters a to z or A to Z including diacritics and ligatures (provided Latin-1 codepage is active)
	%p	special characters, e.g. , . : ; - + * ~ ? ! # _ ( ) [ ] { } " '
	%P	anything not representing special characters
	%s	spaces including \t, \n, and \r
	%S	anything not matching spaces including \t, \n, and \r
	%u	upper-case letters
	%U	anything not representing upper-case letters
	%v	upper and lower-case vowels including y and Y
	%V	anything not representing upper and lower-case vowels including y and Y
	%w	alphanumeric characters a to z, A to Z, and 0 to 9
	%W	anything not matching the class %w
	%x	hexadecimal digits 0 to 9, A to F, and a to f
	%X	anything not matching the class %x
	%z	an embedded zero, i.e. \0.
	%Z	anything not matching an embedded zero

Modifiers		
	+	one or more occurrence
	*	zero or more occurrence, returning the largest match
	−	zero or more occurrence, returning the smallest match
	?	zero or one occurrence

Table 9: Character classes and modifiers

---

[6] Based on: `Programming in Lua`, 2nd edition, by Roberto Ierusalimschy, lua.org, pages 180f.

4.8 Boolean Expressions

Agena supports the logical values **true** and **false**, also called \`booleans\`. Any condition, e.g. a < b, results to one of these logical values. They are often used to tell a programme which statements to execute and thus which statements not to execute.

Boolean expressions mostly result to the Boolean values **true** or **false**. Boolean expressions are created by:

- relational operators (>, <, =, ==, ~=, ~<>, <=, >=, <>),
- logical names: **true**, **false**, **fail**, and **null**,
- **in**, **subset**, **xsubset**, and various functions.

Agena supports the following relational operators:

Operator	Description	Example
<	less than	`1 < 2`
>	greater than	`2 > 1`
<=	less than or equals	`1 <= 2`
>=	greater than or equals	`2 >= 1`
=	equals	`1 = 1`
==	strict equality for structures[7]	`[1] == [1]` `1 == 1`
~=, ~<>	approximate equality/inequality for real and complex numbers, and structures	`1 ~= 1` `[1] ~<> [1]`
<>	not equals	`1 <> 2`

Table 10: Relational operators

The logical operators **and**, **or**, **nand**, **nor**, **xor**, and **xnor** behave a little bit differently: They consider anything except **false**, **fail**, and **null** as true, and false otherwise. They return either the first or second operand, which can be any data - not just **true** or **false** - subject to the following rules:

Operator	Description	Examples

and	Returns its first operand if it is or evaluates to **false**, **fail** or **null**, otherwise returns its second operand.	`true and 1 » 1`
		`false and 1 » false`
		`true and false » false`
		`false and true » false`

or	Returns its first operand if it is not or does not evaluate to **false**, **fail**, or **null**, otherwise it returns its second operand.	`true or true » true`
		`true or false » true`
		`2 or true » 2`
		`null or 2 » 2`

xor	With booleans: Returns the first operand if the second one evaluates or is **false**, **fail**, or **null**. It returns the second operand if the first operand evaluates to	`true xor false» true`
		`true xor true » false`
		`false xor true » true`
		`1 xor null » 1`
		`1 xor 2 » 2`

---

[7] See Chapter 4.9.3.

Operator	Description	Examples
	false, **fail**, or **null** and if the second operand is neither **false**, **fail** nor **null**. With non-booleans: returns the first operand if the second operand evaluates to **null**, otherwise the second operand will be returned.	

not	Turns a true expression to **false** and vice versa.	`not true » false`
		`not false » true`
		`not 1 » false`
		`not null » true`

nand	Returns **true** if at least one operand is **false**, otherwise returns **false**.	`true nand false » true` `1 nand null » true`
nor	Returns **true** if both operands are **false**, and **false** otherwise.	`false nor false » true`
xnor	Returns **true** if both Boolean operands are the same (where **false** and **fail** are considered equal), and **false** otherwise.	`false xnor false » true`
implies	Returns **false** if the first operand is **true** and the second is **false**; otherwise returns **true**.	`false implies false` `» true`

Table 11: Logical operators

As expected, you can assign Boolean expressions to names

```
> cond := 1 < 2:
true

> cond := 1 < 2 or 1 > 2 and 1 = 1:
true
```

or use them in **if** statements, described in Chapter 5.

In many situations, the **null** value can be used synonymously for **false**.

The additional Boolean constant **fail** can be used to denote an error. With Boolean operators (**and**, **or**, **not**), **fail** behaves like the **false** constant, e.g. not(fail) = **true**, but remember that **fail** is always unlike **false**, i.e. the expression **fail** = **false** results to **false**.

true, **false**, and **fail** are of type **boolean**. **null**, however, has its own type: the string '**null**'.

The **and** as well as **or** operators only evaluate their second argument if necessary, called short-circuit evaluation. Thus, the following statement does not issue an error:

```
> a := null

> if a :: number and a > 0 then print(ln(a)) fi
```

They are also handy to define defaults for unassigned names:

```
> a := null

> a := a or 0

> a:
0
```

You can add, subtract, multiply, divide and exponentiate numbers with **true** or **false**, where **true** in this context represents number 1 and **false** or **fail** number 0. Thus, for example, the expressions `abs(x > 0)*x` and `(x > 0)*x` are equivalent expressions. You can even apply the four basic arithmetic operations on two booleans if deemed necessary.

4.9 Tables

Tables are used to represent more complex data structures. Tables consist of zero, one or more key-value pairs: the key referencing to the position of the value in the table, and the value the data itself.

Keys and values can be numbers, strings, and any other data type except **null**.

Here is a first example: Suppose you want to create a table with the following meteorological data recorded by Viking Lander 1 which touched down on Mars in 1976:

Sol	Pressure in mb	Temperature in °C
1.02	7.71	-78.28
1.06	7.70	-81.10
1.10	7.70	-82.96

```
> VL1 := [
>    1.02 ~ [7.71, -78.28],
>    1.06 ~ [7.70, -81.10],
>    1.10 ~ [7.70, -82.96]
> ];
```

To get the data of Sol 1.02 (the Martian day #1.2) input:

```
> VL1[1.02]:
[7.71, -78.28]
```

Tables may be empty, or include other tables - even nested ones.

You can control how tables are printed at the console in two ways: If the setting `environ.kernel('longtable')` is **true** (e.g. by entering the statement `environ.kernel(longtable = true)`, then each key~value pair is printed at a separate line. If the setting `environ.kernel('longtable')` is **false**, all key~value pairs will be printed in one consecutive line, as in the example above. Also, you can define your own printing function that tells the interpreter how to print a table (or

other structures). See Appendix A5 for further information on how to do this and other settings.

Stripped down versions of tables are sets, sequences and registers which are described later. Most operations on tables introduced in this chapter are also applicable to them.

4.9.1 Arrays

Agena features two types of tables, the simplest one being the *array*. Arrays are created by putting their values in square brackets:

$$\textbf{[[} \textit{value}_1 \textbf{ [, } \textit{value}_2, \cdots \textbf{]]]}$$

```
> A := [4, 5, 6]:
[4, 5, 6]
```

The table *values* are 4, 5, and 6; the numbers 1, 2, and 3 are the corresponding *keys* or *indices* of table `A`, with key 1 referencing value 4, key 2 referencing value 5, etc. With arrays, the indices always start with 1 and count upwards sequentially. The keys are always integral, so `A` in this example is an array whereas table `VL1` in the last chapter is not.

To determine a table value, enter the name of the table followed by the respective index in square brackets:

$$\textit{tablename}\textbf{[}\textit{key}\textbf{]}$$

```
> A[1]:
4
```

Instead of using constants to index a table, you may also compute an index both in table assignments or queries. The following selects the middle element of `A`:

```
> l, r := 1, size A:
1        3

> A[(l+r)\2]:
5
```

If a table contains other tables, you may get their values by passing the respective keys in consecutive order. The two forms are equivalent:

$$\textit{tablename}\textbf{[}\textit{key}_1\textbf{][}\textit{key}_2\textbf{][}\cdots\textbf{]}$$
$$\textit{tablename}\textbf{[}\textit{key}_1\textbf{, }\textit{key}_2\textbf{, }\cdots\textbf{]}$$

```
> A := [[3, 4]]:
[[3, 4]]
```

The following call refers to the complete inner table which is at index 1 of the outer table:

```
> A[1]:
[3, 4]
```

The next call returns the second element of the inner table.

```
> A[1][2], A[1, 2]:
4      4
```

Tables may be nested:

```
> A := [4, [5, [6]]]:
[4, [5, [6]]]
```

To get the number `6`, enter the position of the inner table `[5, [6]]` as the first index, the position of the inner table `[6]` as the second index, and the position of the desired entry as the third index:

```
> A[2, 2, 1]:
6
```

With tables that contain other tables, you might get an error if you use an index that does not refer to one of these tables:

```
> A[1][0]:
Error in stdin, at line 1:
   attempt to index field `?` (a number value)
```

Here `A[1]` returns the number `4`, so the subsequent indexing attempt with `4[0]` is an invalid expression. You may use the **getentry** function to avoid error messages:

```
> getentry(A, 1, 0):
null
```

Similarly, the **..** operator allows to index tables even if its left-hand side operand evaluates to **null**. In this case, **null** will be returned, as well, with no error issued. It is twice as fast as **getentry**.

```
> create table A;

> A.b:
null

> A.b.c:
Error in stdin, at line 1:
   attempt to index field `b` (a null value)

> A..b..c:
null
```

A generalisation of the **..** table field separator are curly braces.

```
> create table A;

> A[1]:
null

> A[1, 2]:
Error in stdin, at line 1:
   attempt to index field `?` (a null value)

> A{1, 2}:
null
```

Sublists of table arrays can be determined with the following syntax:

$$\textit{tablename}[\textit{m} \textbf{ to } \textit{n}]$$

Agena returns all values from and including index position *m* to *n*, with *m* and *n* negative or positive integers or 0. If there are no values between *m* and *n*, an empty list will be returned. Table values with non-integral keys are ignored.

```
> A := [10, 20, 30, 40]

> A[2 to 3]:
[2 ~ 20, 3 ~ 30]
```

Tables can contain no values at all. In this case they are called *empty tables* with values to be inserted later in a session. There are two forms to create empty tables.

$$\textbf{create table } \textit{name}_1 \, [, \textbf{ table } \textit{name}_2, \; \cdots]$$
$$\textit{name}_1 := \textbf{[ ]}$$

```
> create table B;
```

creates the empty table B,

```
> B := [];
```

does exactly the same.

You may add a value to a table by assigning the value to an indexed table name:

```
> B[1] := 'a';

> B:
[a]
```

Alternatively, the **insert** statement always appends values to the end of a table[8]:

> insert value_1 [, value_2, · · ·] into name

```
> insert 'b' into B;

> B:
[a, b]
```

To delete a specific key~value pair, assign **null** to the indexed table name:

```
> B[1] := null;

> B:
[2 ~ b]
```

The **delete** [9]statement works a little bit differently and removes all occurrences of a value from a table.

> delete value_1 [, value_2, · · ·] from name

```
> insert 'b' into B;

> delete 'b' from B;

> B:
[]
```

In both cases, deletion of values leaves `holes` in a table, which are **null** values between other non-**null** values:

```
> B := [1, 2, 2, 3]

> delete 2 from B

> B:
[1 ~ 1, 4 ~ 3]
```

There exists a special sizing option with the **create table** statement which besides creating an empty table also sets the default number of entries. Thus you may gain some speed if you perform a large number of subsequent table insertions, since with each insertion, Agena checks whether there is enough space to accommodate further elements and allocates more space if necessary, which creates some overhead. The sizing option reserves memory for the given number of elements in advance, so there is no need for Agena to subsequently enlarge the table until the given default size has been exceeded.

---

[8] The **insert** statement cannot be applied on weak tables. See Chapter 6 for further information on this variant.

[9] dito.

Arrays with a predefined number of entries are created according to the following syntax:

create table *name₁***(***size₁***)** [, **table** *name₂***(***size₂***)**, · · ·]

When assigning entries to the table, you will save at least 1/3 of computation time if you know the size of the table in advance and initialise the table accordingly. If you want to insert more values later, then this will be no problem. Agena automatically enlarges the table beyond its initial size if needed.

```
> create table a(5);
```

```
> create table a, table b(5);
```

4.9.2 Dictionaries

Another form of a table is the *dictionary* with any kind of data - not only positive integers - as indices:

Dictionaries are created by explicitly passing key-value pairs with the respective keys and values separated by tildes, which is the difference to arrays:

[[*key₁* ~ *value₁* [, *key₂* ~ *value₂*, · · ·]] **]**

```
> A := [1 ~ 4, 2 ~ 5, 3 ~ 6]:
[1 ~ 4, 2 ~ 5, 3 ~ 6]

> B := [abs('þ') ~ 'th']:
[231 ~ th]
```

Here is another example with strings as keys:

```
> dic := ['donald' ~ 'duck', 'mickey' ~ 'mouse'];

> dic:
[mickey ~ mouse, donald ~ duck]
```

As you see in this example, Agena internally stores the key-value pairs of a dictionary in an arbitrary order.

As with arrays, indexed names are used to access the corresponding values stored to dictionaries.

```
> dic['donald']:
duck
```

If you use strings as keys, a short form is:

```
> dic.donald:
duck
```

Further entries can be added with assignments such as:

```
> dic['minney'] := 'mouse';
```

which is the equivalent to

```
> dic.minney := 'mouse';
```

With string indices, an alternative to putting keys in quotes with the tilde syntax is:

$$\boxed{\textbf{[} \ [name_1 = value_1 \ [, \ name_2 = value_2, \ \cdots]] \ \textbf{]}}$$

Hence,

```
> dic := ['donald' ~ 'duck', 'mickey' ~ 'mouse'];
```

and

```
> dic := [donald = 'duck', mickey = 'mouse'];
```

are equal. You can also mix tilde (~) and equals (=) assignments:

```
> dic := [donald = 'duck', mickey ~ 'mouse'];
```

If you want to enter the result of a Boolean equality check into a table, use the ==
token instead of the = sign:

```
> value := 1

> [value == 1, value <> 1]:
[true, false]
```

Dictionaries with an initial number of entries are declared like this:

$$\boxed{\textbf{create dict } name_1\textbf{(}size_1\textbf{)} \ [, \ \textbf{dict } name_2\textbf{(}size_2\textbf{)}, \ \cdots]}$$

You may mix declarations for arrays and dictionaries, so the general syntax is:

$$\boxed{\textbf{create } \{\textbf{table} \mid \textbf{dict}\} \ name_1[\textbf{(}size_1\textbf{)}] \ [, \ \{\textbf{table} \mid \textbf{dict}\} \ name_2[\textbf{(}size_2\textbf{)}], \ \cdots]}$$

Technically, tables consist of an array and a hash part. The array part usually stores
all the elements in an array, the hash part the values of a dictionary. You can both
pre-allocate the array and hash part of a table at once:

$$\boxed{\textbf{create table } name_1\textbf{(}\textbf{\textit{array}}size_1\textbf{, } \textbf{\textit{hashsize}}_1\textbf{) [}, \cdots\textbf{]}}$$

4.9.3 Table, Set and Sequence Operators

Agena features some built-in table, set and sequence operators which are described below. A `structure` in this context is a table, set or sequence.

Name	Return	Function
c **in** A	Boolean	Checks whether the structure A contains the given value c.
filled A	Boolean	Determines whether a structure contains at least one value. If so, it returns **true**, else **false**.
empty A	Boolean	Checks whether a structure is empty.
A = B	Boolean	Checks whether two structures A, B contain the same values regardless of the number of their occurrence and order; if B is a reference to A, then the result is also **true**.
A <> B	Boolean	Checks whether two structures A, B do not contain the same values regardless of the number of their occurrence or order; if B is a reference to A, then the result is **false**.
A == B	Boolean	Checks whether two structures A, B contain the same number of elements and whether all key~value pairs in tables A, B or entries in the sets, sequences or registers are the same; if B is a reference to A, then the result is **true**.
not(A == B)	Boolean	The negation of A == B.
A ~= B	Boolean	Like ==, but checks the respective elements for approximate equality. Use **environ.kernel/eps** to change the setting for the accuracy threshold.
not(A ~= B)	Boolean	The negation of A ~= B.
A **subset** B	Boolean	Checks whether the values in structure A are also values in B regardless of the number of their occurrence. The operator also returns **true** if A = B.
A **xsubset** B	Boolean	Checks whether the values in structure A are also values in B. Contrary to **subset**, the operator returns **false** if A = B.
A **union** B	table, set, seq, reg	Concatenates two tables, or two sets, or two sequences or registers A, B simply by copying all its elements - even if they occur multiple times - to a new structure. With sets, all items in the resulting set will be unique, i.e. they will not appear multiple times.
A **intersect** B	table, set, seq, reg	Returns all values in two structures A, B that are included both in A and in B and returns them in a new structure.

Name	Return	Function
A **minus** B	table, set, seq, reg	Returns all the values in A that are not in B as a new structure.
copy A	table, set, seq, reg	Creates a deep copy of structure A, i.e. if A includes other tables, sets, pairs, sequences or registers, copies of these structures are created, too.
join A	string	Concatenates all strings in the table, sequence or register A.
size A	number	Returns the size of a table A, i.e. the actual number of key~value pairs in A. With sets, sequences and registers, the number of items will be returned.
sort(A**)**	table, seq, reg	This function sorts table, sequence or register A in ascending order. It directly operates on A, so it is destructive. With tables, the function has no effect on values that have non-integer keys. Note that **sort** is not an operator, so you must put the argument in brackets. Please also see Chapter 7 for its derivatives: **sorted**, **skycrane.sorted**, **stats.issorted**, and **stats.sorted**.
unique A	table, seq, reg	Removes multiple occurrences of the same value and returns the result in a new structure. With tables, also removes all holes (`missing keys`) by reshuffling its elements. This operator is not applicable to sets, since they are already unique.
sumup A	number	Sums up all numeric table, sequence or register values. If the structure is empty or contains no numeric values, **null** will be returned. Sets are not supported.
qsumup A	number	Raises each value in a table, sequence or register to the power of 2 and sums up these powers. If the structure is empty or contains no numeric values, **null** will be returned. Sets are not supported.
f @ A	table, seq, set, reg	Maps a function f on all elements of structure A.
f $ A	table, set, seq, reg	Selects all elements of a structure A that satisfy a condition given by function f.
f $$ A	Boolean	Checks whether at least one element in A satisfies the condition checked by function f.

Table 12: Table, set, and sequence operators

Here are some examples - try them with sets, sequences and registers, as well:

The **union** operator concatenates two tables simply by copying all its elements - even if they occur multiple times.

```
> ['a', 'b', 'c'] union ['a', 'd']:
[a, b, c, a, d]
```

intersect returns all values that are part of both tables as a new table.

```
> ['a', 'b', 'c'] intersect ['a', 'd']:
[a]
```

If a value appears multiple times in the structure at the left hand side of the operator, it is written the same number of times to the resulting structure.

minus returns all the elements that appear in the table on the left hand side of this operator that are not members of the right side table.

```
> ['a', 'b', 'c'] minus ['a', 'd']:
[b, c]
```

If a value appears multiple times in the structure at the left hand side of the operator, it is written the same number of times to the resulting structure.

The **unique** function

- removes all holes (`missing keys`) in a table,
- removes multiple occurrences of the same value.

and returns the result in a new table. The original table is *not* overwritten. In the following example, there is a hole at index 2 and the value 'a' appears twice.

```
> unique [1 ~ 'a', 3 ~ 'a', 4 ~ 'b']:
[b, a]
```

You can search a table for a specific value with the **in** operator. It returns **true** if the value has been found, or **false**, if the element is not part of the table. Examples:

```
> 'a' in ['a', 'b', 'c']:
```

returns **true**.

```
> 1 in ['a', 'b', 'c']:
```

returns **false**. Remember that **in** only checks the *values* of a table, not its keys.

4.9.4 Table Functions

Agena has a number of functions that work on tables (and sequences and registers), for instance:

Function	Description	Further detail
map(*f*, *o*) **map**(*f*, g)	Maps a function *f* onto all elements of structure *o*, or produces the function composition f @ g.	*f* may be an anonymous function, as well. See also **zip** in Chapter 8.
purge(*o*, *key*)	Removes index *key* and its corresponding value from *o*.	All elements to the right are shifted down, so that no holes are created.
put(*o*, *key*, *value*)	Inserts a *key* ~ *value* pair into structure o.	The original element at position *key* and all other elements are shifted up one place.
select(*f*, *o*)	Returns all the elements that satisfy the Boolean condition given by function *f*.	*f* may be also an anonymous function. The **remove** function conducts the opposite operation.
subs(*o*, *x:v*)	Substitutes all occurrences of value *x* in *o* with value *v*.	
binsearch(*o*, *i*)	Performs a binary search in a table.	With large tables, the function is much faster than the **in** operator.

Table 13: Basic table library procedures

The **map** function is quite handy to apply a function with one or more arguments to all elements of a structure in one stroke:

```
> map(<< x -> x^2 >>, [1, 2, 3]):
[1, 4, 9]
```

The @ operator also maps a function on all elements of a structure. Contrary to **map**, it accepts univariate functions only, but is faster:

```
> << x -> x^2 >> @ [1, 2, 3]:
[1, 4, 9]
```

Likewise, the faster **$** operator selects all the elements of a structure that satisfy a condition checked by a univariate function.

```
> << x -> x > 1 >> $ [1, 2, 3]:
[2, 3]
```

Suppose we want to add a new entry 10 at position 3 of table c[10]:

```
> C := [1, 2, 3, 4]

> put(C, 3, 10)
```

---

[10] **put** and **purge** have to shift elements up or down, drawing performance. You may use the llist package to conduct these kinds of operations much faster in case of a large number of insertions or deletions.

```
> C:
[1, 2, 10, 3, 4]
```

Now we remove this new entry 10 at position 3 again:

```
> purge(C, 3)

> C:
[1, 2, 3, 4]
```

Determine all elements in c that are even:

```
> select(<< x -> even(x) >>, C):
[2 ~ 2, 4 ~ 4]
```

Or return all elements not even:

```
> remove(<< x -> even(x) >>, C):
[1 ~ 1, 3 ~ 3]
```

Note that **remove** and **select** do not alter the original structure passed as the second argument.

zip zips together two tables by applying a function to each of its respective elements.

```
> C:
[1, 2, 3, 4]

> zip(<< (x, y) -> x + y >>, C, [10, 20, 30, 40]):
[11, 22, 33, 44]
```

For other functions, have a look at Part II of this manual and the Agena Quick Reference Excel sheet.

4.9.5 Table References

If you assign a table to a variable, only a reference to the table is stored in the variable. This means that if we have a table

```
> A := [1, 2];
```

assigning

```
> B := A;
```

does not copy the contents of A to B, but only the address of the same memory area which holds table [1, 2], hence:

```
> insert 3 into A;
```

```
> A:
[1, 2, 3]
```

also yields:

```
> B:
[1, 2, 3]
```

Use **copy** to create a true copy of the contents of a table. If the table contains other structures, copies of these structures are also made (so-called `deep copies`). Thus **copy** returns a new table without any reference to the original one.

```
> B := copy(A);

> insert 4 into A;

> B:
[1, 2, 3]
```

With structures such as tables, sets, pairs, sequences or registers, all names to the left of an `->` token will point to the very same structure to its right.

```
> A, B -> []

> A[1] := 1

> B:
[1]
```

Tables can also directly or indirectly contain themselves, in which case they are also called `cycles`. Just some few examples:

```
> A := []

> A := [A, A]

> A:
[[], []]

> A.A := A

> A:
[1 ~ [], 2 ~ [], A ~ circum_table(0236A460)]
```

4.9.6 Unpacking Tables by Name

There is syntactic sugar for the assignment statement to unpack named values, i.e. data indexed with string keys, from tables using the **in** keyword:

$$key_1 \, [, \, key_2, \, \cdots] \textbf{ in } tablename$$

is equal to

$$key_1 \;[, key_2, \; \cdots] := tablename.key_1 \;[, tablename.key_2, \; \cdots]$$

A short example may suffice:

```
> zips := [duedo   = 40210:40629,
>          bonn    = 53111:53229,
>          cologne = 50667:51149];

> duedo, bonn in zips

> duedo, bonn, cologne:
40210:40629      53111:53229      null
```

The **local** statement, see Chapter 6.2, supports this sugar, as well. Read also Chapter 5.2.12 for a variant implemented available in the **with** statement.

4.9.7 Defining Multiple Constants Easily

The // ... \\ constructor allows to define a table of constant numbers and/or strings the simple way: items may not be separated by commas, and strings do not need to be put in quotes as long as they satisfy the criteria for valid variable names: names starting with a hyphen or letter, including diacritics - and keywords such like **while**, **sqrt**, etc. do not have to be passed in quotes. Records are supported as well. Expressions like `sin(0)` etc. are *not* parsed and rejected. Example:

```
> a := // 0~0 1 2 3 zero one two three '2and3' sqrt ~ while \\:
[0 ~ 0, 1 ~ 1, 2 ~ 2, 3 ~ 3, 4 ~ zero, 5 ~ one, 6 ~ two, 7 ~ three,
8 ~ 2and3, sqrt ~ while]
```

4.10 Sets

Sets are collections of unique items: numbers, strings, and any other data except **null**. Their syntax is:

$$\{\; [\; item_1 \;[, item_2, \; \cdots]\;]\;\}$$

Thus, they are equivalent to Cantor sets: An item is stored only once.

```
> A := {1, 1, 2, 2}:
{1, 2}
```

Besides being commonly used in mathematical applications, they are also useful to hold word lists where it only matters to see whether an element is part of a list or not:

```
> colours := {'red', 'green', 'blue'};
```

If you want to check whether the colour red is part of the set colours, just index it as follows:

$$setname[item]$$

If an element is stored to a set, Agena returns **true**:

```
> colours['red']:
true
```

If an item is not in the given set, the return is **false**. Note that we can use the same short form for indexing values (without quotes) as can be done with tables.

```
> colours.yellow:
false
```

If you want to add or delete items to or from a set, use the **insert** and **delete** statements. The standard assignment statement `setname[key] := value` is also supported.

insert *item₁* [, *item₂*, · · ·] **into** *name*

delete *item₁* [, *item₂*, · · ·] **from** *name*

```
> insert 'yellow' into colours;
```

The **in** operator checks whether an item is part of a set - it is an alternative to the indexing method explained above, and returns **true** or **false**, too.

```
> 'yellow' in colours:
true
```

The data type of a set is **set**.

```
> type(colours):
set
```

You may predefine sets with a given number of entries according to the following syntax:

create set *name₁* [(*size₁*)] [, **set** *name₂* [(*size₂*)], · · ·]

When assigning items later, you will save at least 90 % of computation time if you know the size of the set in advance and initialise it with the maximum number of future entries as explained above. More items than stated at initialisation can be

entered anytime, since Agena automatically enlarges the respective set accordingly and will also reserves space for additional entries.

Sets are useful in situations where the number of occurrence of a specific item or its position does not concern. Compared to tables, sets consume around 40 % less memory, and operations with them are 10 % to 33 % faster than the corresponding table operations.

Specifically, the more items you want to store, the faster operations will be compared to tables.

Note that if you assign a set to a variable, only a reference to the set is stored in the variable. Thus in a statement like `A := {}; B := A`, A and B point to the same set. Use the **copy** function if you want to create `independent` sets.

Sets can also include themselves, just an example:

```
> A := {}
```

```
> A := {A, A}:
{{{}}}
```

If you want to know the number of occurrence of a unique element in a distribution, the **bags** package might be of interest, see Chapter 10.8.

The following operators operate on sets:

Name	Return	Function
c **in** A	Boolean	Checks whether the set A contains the given value c.
filled A	Boolean	Determines whether a set contains at least one value. If so, it returns **true**, else **false**.
empty A	Boolean	Checks whether a set is empty.
A = B	Boolean	Checks whether two sets A, B contain the same values; if B is a reference to A, then the result is also **true**.
A <> B	Boolean	Checks whether two sets A, B do not contain the same values; if B is a reference to A, then the result is **false**.
A == B	Boolean	Same as =.
A **subset** B	Boolean	Checks whether the values in set A are also values in B. The operator also returns **true** if A = B.
A **xsubset** B	Boolean	Checks whether the values in set A are also values in B. Contrary to **subset**, the operator returns **false** if A = B.
A **union** B	set	Concatenates two sets A, B simply by copying all its elements to a new set. All items in the resulting set will be unique, i.e. they will not appear multiple times.
A **intersect** B	set	Returns all values in two sets A, B that are included both in A and in B as a new set.
A **minus** B	set	Returns all the values in A that are not in B as a new set.

Name	Return	Function
copy A	set	Creates a deep copy of the set A, i.e. if A includes other tables, sets, pairs, sequences or registers, copies of these structures are built, too.
size A	number	Returns the size of a set A, i.e. the actual number of elements in A.
f @ A	set	Maps a function f on all elements of a set A.
f $ A	set	Selects all elements in A that satisfy a given condition checked by function f.
f $$ A	Boolean	Checks the elements in A whether at least one satisfies a given condition checked by function f.

Table 14: Set operators

4.11 Sequences

Besides storing values in tables or sets, Agena also features the sequence, an object which can hold any number of items except **null**. You may sequentially add items and delete items from it. Compared to tables, insertion and deletion are twice as fast with sequences. Contrary to all other data structures, Agena automatically frees the memory occupied by a sequence if you remove values from it[11].

Sequences store items in sequential order. As with tables, an item may be included multiple times. Sequences are usually indexed with positive integers in the same fashion as table arrays are, starting at index 1. If you pass a negative index n, then the $|n|$-th value from the right end, i.e. the top of the sequence is determined. Non-integral indices are not allowed. As with tables, you can compute the index in assignments or queries.

Suppose we want to define a sequence of two values. You may create it using the **seq** operator.

> **seq(** [*item$_1$* [, *item$_2$*, $\cdots$]] **)**

```
> a := seq(0, 1, 2, 3);

> a:
seq(0, 1, 2, 3)
```

You can access the items the usual way:

> *seqname[**index**]*

```
> a[1]:
0
```

---

[11]You can turn off this feature by issuing: `environ.kernel(seqautoshrink = false)`.

```
> a[2]:
1
```

If the index is larger than the current size of the sequence, an error will be returned[12].

```
> a[5]:
Error, line 1: index out of range
```

Sublists of sequences can be determined with the following syntax:

$$seqname[m \text{ to } n]$$

Agena returns all values from and including index position *m* to *n*, with *m* and *n* positive or negative integers. In case of a non-existing key, an error will be issued.

```
> a[2 to 3]:
seq(1, 2)
```

The way Agena outputs sequences can be changed by using the **settype** function.

In general, the **settype** function allows you to set a user-defined subtype for a sequence, set, table or pair.

```
> a := seq(0, 1);

> settype(a, 'duo');

> a:
duo(0, 1)
```

The **gettype** function returns the new type you defined above as a string:

```
> gettype(a):
duo
```

If no user-defined type has been set, **gettype** returns **null**.

Once the type of a sequence has been set, the **typeof** operator also returns this user-defined sequence type and will not return `'sequence'`.

```
> typeof(a), gettype(a):
duo    duo
```

This allows you to programme special operations only applicable to certain types of sequences.

The **::** and **:-** operators can check user-defined types. Just pass the name of your type as a string:

---

[12] The error message can be avoided by defining an appropriate metamethod.

```
> a :: 'duo':
true

> a :- 'duo':
false
```

Note that if a user defined-type has been given, the check for a basic type with the **::** and **:-** operators will return also return **true** or **false**.

```
> a :: sequence:
true

> a :- sequence:
false
```

A user-defined type can be deleted by passing **null** as a second argument to **settype**.

```
> settype(a, null);

> typeof(a):
sequence
```

The **create sequence** statement creates an empty sequence and optionally allows to allocate enough memory in advance to hold a given number of elements (which can be inserted later). Agena automatically will extend the sequence, if the predetermined number of items is exceeded. The **sequence** and **seq** keywords are synonyms.

> **create sequence** $name_1$ [, **seq** $name_2$, $\cdots$]
> **create sequence** $name_1(size_1)$ [, **seq** $name_2(size_2)$, $\cdots$]

Items can be added only sequentially. You may use the **insert** statement for this or the conventional indexing method.

```
> create sequence a(4);

> insert 1 into a;

> a[2] := 2;

> a:
seq(1, 2)
```

Note that if the index is larger than the number of items stored to it plus 1, Agena returns an error in assignment statements, since `holes` in a sequence are not allowed. The next free position in `a` is at index 3, however a larger index is chosen in the next example.

```
> a[4] := 4
Error, line 1: index out of range

> a[3] := 3
```

Items can be deleted by setting their index position to **null**, or by applying **delete**, i.e. stating which items - not index positions - shall be removed. Note that all items to the right of the value deleted are shifted to the left, thus their indices will change.

```
> a[1] := null

> a:
seq(2, 3)
> delete 2, 3 from a

> a:
seq()
```

Thus concerning the **insert** and **delete** statements, we have the following familiar syntax:

> **insert** *item₁* [, *item₂*, · · ·] **into** *name*
>
> **delete** *item₁* [, *item₂*, · · ·] **from** *name*

If you assign a sequence to a variable, only a reference to the sequence is stored in the variable. Thus sequences behave the same way as tables and sets do, i.e. in a statement like `A := seq(); B := A`, `A` and `B` point to the same sequence in memory. Use the **copy** function if you want to create `independent` sequences.

```
> A := seq()

> B := A

> A[1] := 10

> B:
seq(10)
```

As with tables and sets, sequences can also reference to themselves:

```
> A := seq()

> A[1] := A

> A[2] := A

> A:
seq(circum_sequence(01E647D8), circum_sequence(01E647D8))
```

The following operators, functions, and statements operate on sequences:

Name	Description	Example
=	Equality check the Cantor way	`a = b`
==	Strict equality check	`a == b`
~=	approximate equality check	`a ~= b`
<>	Inequality check the Cantor way	`a <> b`
::	Type check operator	`a :: sequence` `a :: 'usertype'`

Name	Description	Example
:-	Negation of type check operation	`a :- sequence` `a :- 'usertype'`
@	Maps a function on all elements of a sequence.	`f @ a`
$	Selects all elements of A that satisfy a given condition.	
$$	Checks whether at least one element in A satisfies a condition.	`f $$ a`
insert	Inserts one or more elements.	`insert 1 into a`
delete	Deletes one or more elements.	`delete 0, 1` `    from a`
bottom	Returns the item with key 1.	`bottom a`
top	Returns the item with the largest key.	`top a`
pop	as an operator works like top but also removes the item from the sequence	`pop a`
copy	Creates an exact copy of a sequence; deep copying is supported so that structures inside sequences are properly treated.	`copy a`
filled	Checks whether a sequence has at least one item.	`filled a`
empty	Checks whether a sequence is empty.	`empty a`
getentry	Returns entries without issuing an error if a given index does not exist.	`getentry(a, 1, 3)`
in	Checks whether an element is stored in the sequence, and returns **true** or **false**. See also **binsearch**.	`0 in seq(1, 0)`
join	Concatenates all strings in a sequence in sequential order.	`join(a)`
pop	Pops the first or the last element from a sequence.	`pop bottom from a` `pop top from a`
size	Returns the current number of items.	`size a`
sort	Sorts a sequence in place. Please also see Chapter 7 for its derivatives: **sorted**, **skycrane.sorted**, **stats.issorted**, and **stats.sorted**.	`sort(a)`
type	Returns the general type of a sequence, i.e. **sequence**.	`type a`
typeof	Returns the user-defined type of a sequence, or the basic type if no special type has been defined.	`typeof a`
unique	Reduces multiple occurrences of an item in a sequence to just one.	`unique a`
unpack	Unpacks a sequence. See **unpack** in Chapter 8.	`unpack(a)`
nseq	Creates a new sequence and fills it with values	`nseq(<< x -> x` `   >>, 1, 10)`
map	Maps a function on all elements of a sequence.	`map(<< x -> x^2` `>>, seq(1, 2, 3))`

Name	Description	Example
zip	Zips together two sequences by applying a function to each of its respective elements.	`zip(<< x, y -> x + y >>, seq(1, 2), seq(3, 4))`
intersect	Searches all values in one sequence that are also values in the other sequence and returns them in a new sequence.	`seq(1, 2) intersect seq(2, 3)`
minus	Searches all values in one sequence that are not values in the other sequence and returns them as a new sequence.	`seq(1, 2) minus seq(2, 3)`
subset	Checks whether all values in a sequence are included in the other sequence.	`seq(1) subset seq(1, 2)`
union	Concatenates two sequences simply by copying all its elements.	`seq(1, 2) union seq(2, 3)`
settype	Sets a user-defined type for a sequence.	`settype(a, 'duo')`
gettype	Returns a user-defined type for a sequence.	`gettype(a)`
setmeta-table	Assigns a metatable to a sequence.	`setmetatable (a, mtbl)`
getmeta-table	Returns the metatable stored to a sequence.	`getmetatable(a)`

Table 15: Basic sequence operators and functions

For more functions, consult the Agena Quick Reference Excel sheet. Also, you may have a look at the **llist** linked list package presented in Chapter 6.27, if you have to conduct a lot of insertions and/or deletions in a data structure.

The (/ ... \) constructor allows to define a sequence of constant numbers and/or strings the simple way: items may not be separated by commas, and strings do not need to be put in quotes as long as they satisfy the criteria for valid variable names(name starting with a hyphen or letter, including diacritics) or if they are keywords. Expressions like `sin(0)` etc. are rejected.

Example:

```
> a := (/ 0 1 2 3 zero one two three '2and3' while \):
seq(0, 1, 2, 3, zero, one, two, three, 2and3), while]
```

4.12 Stack Programming

Sequences and sometimes table arrays can be used to implement stacks, and besides the **insert**/**into** statement to put an element to the top, an efficient statement is available to remove an item from the bottom or from the top of the stack:

> **pop bottom from** *name*
>
> **pop top from** *name*

Both variants work on tables even if their integer keys are not distributed consecutively.

The **bottom** and **top** operators return the element at the bottom of the stack and the top of the stack, respectively. They both do not delete the element returned from the stack.

```
> stack := seq();

> insert 10, 11, 12 into stack;

> bottom(stack):
10

> top(stack):
12

> pop bottom from stack;

> pop top from stack;

> stack:
seq(11)
```

The **rotate** statement moves each element in a sequence or the array part of a table one position to the bottom (downwards) or to the top (upwards):

> **rotate bottom** *name*
>
> **rotate top** *name*

The element at the bottom or the top is moved to the top or the bottom, respectively.

```
> s := seq(1, 2, 3);

> rotate bottom s;

> s:
seq(2, 3, 1)

> s := seq(1, 2, 3):
seq(1, 2, 3)

> rotate top s;

> s:
seq(3, 1, 2)
```

The **pop** operator - contrary to **top** - both returns the top element of a sequence or register and then removes it from the structure. With tables, it returns the value indexed by the largest integer key and then also removes it.

```
> pop(s):
2

> s:
seq(3, 1)
```

There are two other statements that work on sequences and registers only: The **exchange** statement swaps the two topmost elements, and the **duplicate** statement adds a copy of the current topmost element to the end of the structure.

```
> exchange s

> s:
seq(1, 3)

> duplicate s

> s:
seq(1, 3, 3)
```

You may try to use the **put** function to insert new values in the interior of a stack, shifting up other values to open space, and **purge** to delete values in the interior of a stack.

See also Chapter 14.6 for the six built-in number and character stacks.

4.13 More on the create Statement

You cannot only initialise any table arrays with the **create** statement, but also dictionaries, sets, and sequences with only one call and in random order, so the following statement is valid:

```
> create table a, dict b(10), set c, sequence d(100), table e(10);

> a, b, c, d, e:
[]        []        {}        seq()    []
```

4.14 Pairs

The structure which holds exactly two values of any type (including **null** and other pairs) is the *pair*. A pair cannot hold less or more values, but its values can be changed. Conceived originally to allow passing options in a more flexible way to functions, it is defined with the colon operator:

$$item_1 : item_2$$

```
> p := 1:2
```

```
> p:
1:2
```

The **left** and **right** operators provide read access to its left and right operands; the standard indexing method using indexed names is supported, as well:

> **left** [(] *pair* [)]
> **right** [(] *pair* [)]

```
> left(p), p[1]:
1       1

> right p, p[2]:
2       2
```

An operand of an existing pair can be changed by assigning a new value to an indexed name, where the left operand is indexed with number 1, and the right operand with number 2:

```
> p[1] := 2;

> p[2] := 3;
```

You can compute the index as long as the result evaluates to the integers 1 or 2, as well.

As with sequences, you may define user-defined types for pairs with the **settype** function which also changes the way pairs are output.

```
> typeof(p):
pair

> settype(p, 'duo');

> p:
duo(2, 3)

> typeof(p):
duo

> gettype(p):
duo

> p :: pair:
true

> p :: 'duo':
true
```

The only other operators besides **left** and **right** that work on pairs are equality (=, ==, ~=), inequality (<>, ~<>), **::**, **:-**, **type**, **typeof**, and **in**.

```
> p = 3:2:
false
```

With pairs consisting of numbers, the **in** operator checks whether a left-hand argument number is part of a closed numeric interval given by the given right-hand argument pair.

```
> 2 in 0:10:
true

> 's' in 0:10:
fail
```

As with all other structures, if you assign a pair to a variable, only a reference to the pair is stored in the variable. Thus in a statement like `A := a:b; B := A`, A and B point to the same pair. Use the **copy** function if you want to create `independent` pairs.

Summary:

Name	Description	Example
=, ==, ~=	Equality checks (mostly same functionality)	`a = b`
<>	Inequality check	`a <> b`
::	Type check operator	`a :: pair` `a :: 'udeftype'`
:-	Negation of type check operation	`a :- pair` `a :- 'udeftype'`
@	Maps a function on each operand.	`f @ a`
copy	Creates an exact copy of a pair; deep copying is supported so that structures inside pairs are properly treated.	`copy a`
in	If the left operand x is a number and if the left and right hand side of the pair a:b are numbers, then the operator checks whether x lies in the closed interval [a, b] and returns **true** or **false**. If at least one value x, a, b is not a number, the operator returns **fail**.	`1.5 in 1:2`
left	Returns the left operand of a pair.	`left(a)`
right	Returns the right operand of a pair.	`right(a)`
type	With pairs, always returns `'pair'`.	`type(a)`
typeof	Returns either the user-defined type of the pair, or the basic type (`'pair'`) if no special type was defined for the pair.	`typeof(a)`
settype	Sets a user-defined type for a pair.	`settype(a, 'duo')`
gettype	Returns the user-defined type of a pair.	`gettype(a)`
setmetatable	Sets a metatable to a pair.	`setmetatable(p, mtbl)`
getmetatable	Returns the metatable stored to a pair.	`getmetatable(p)`

Table 16: Operators and functions applicable to pairs

4.15 Registers

Registers are memory-efficient, fixed-size Agena `sequences` that also store **null**'s. They are not automatically extended if more values have to be added, but can be manually resized.

Registers allow to hide data: by changing the pointer to the top of a register using **registers.settop**, any values stored above (the position of) this pointer can neither be read nor changed by any of Agena's functions and operators. Registers are supported by most of the existing statements, operators and functions. Please also refer to Chapter 6.15 `Sandboxes`.

The concept of the fixed size and the top pointer is key to understanding and working with registers.

By default, the top pointer always refers to the very last element in a register - it is automatically changed only if an element is removed with the **pop top** or **pop bottom** statements, the **pop** operator or the **purge** function.

In general, registers can save memory if you know the precise number of values to be stored, or to be added or removed later, in advance. As such, they behave like C arrays storing any value without provoking faults. With respect to sequences, there usually are no performance gains with most operations - but since registers do not automatically shift elements, they are eight times faster when deleting items.

Let us first create a register with eight items:

```
> a := reg(1, 2, 3, 4, 5, 6, 7, 8):
reg(1, 2, 3, 4, 5, 6, 7, 8)
```

Read the first element:

```
> a[1]:
1
```

Set the first entry to **null** - contrary to other data structures, the size of register is not reduced, and no values are shifted.

```
> a[1] := null;

> a:
reg(null, 2, 3, 4, 5, 6, 7, 8)
```

Now reset the pointer to the top of the register to the fourth element:

```
> registers.settop(a, 4);

> size(a):
4

> a:
reg(null, 2, 3, 4)
```

```
> a[5]:
In stdin at line 1:
   Error: register index 5 out of current range.

Stack traceback:
   stdin, at line 1 in main chunk
```

By changing the position of the top pointer beyond 4, we can read and change the values again:

```
> registers.settop(a, 8);

reg(null, 2, 3, 4, 5, 6, 7, 8)
```

When passing no elements to the **reg** operator, by default a register with sixteen slots is created.

```
> reg():
reg(null, null, null, null, null, null, null, null, null, null, null, null,
null, null, null, null)
```

But you can change this default to another value:

```
> environ.kernel(regsize = 8);

> reg():
reg(null, null, null, null, null, null, null, null)
```

Registers containing **null**'s may issue errors with some functions or operators.

Changing the **size** of a register at runtime is easy:

```
> b := reg('a', 'b', 'c'):
reg(a, b, c)
```

register.extend enlarges a register to the given number of elements.

```
> registers.extend(b, 8);

> b:
reg(a, b, c, null, null, null, null, null)
```

register.reduce shrinks a register to the given number of elements.

```
> registers.reduce(b, 4);

> b:
reg(a, b, c, null)
```

Registers support metamethods and user-defined types. To hide the current size of the register as defined above, we could assign:

```
> size a:
8
```

```
> mt := [
>     '__size' ~ proc(x) is
>         return 0
>     end
> ]

> setmetatable(a, mt);

> size a:
0
```

Name	Description	Example
=	Equality check the Cantor way	`a = b`
==	Strict equality check	`a == b`
~=	Approximate equality check	`a ~= b`
<>	Inequality check the Cantor way	`a <> b`
::	Type check operator	`a :: register`
:-	Negation of type check operation	`a :- register`
@	Maps a function on all elements of a register.	`f @ a`
$	Selects all elements of a that satisfy a given condition.	`f $ a`
$$	Checks whether at least one element satisfies a given condition.	`f $$ a`
insert	Inserts an element at the first position that holds a **null** value.	`insert 0, 1 into a`
delete	Deletes one or more elements and replaces them with **null**.	`delete 0, 1 from a`
bottom	Returns the item with key 1.	`bottom a`
top	Returns the item with the largest key.	`top a`
pop	as an operator works like **top** but also removes the item from the register.	`pop a`
copy	Creates an exact copy of a register; deep copying is supported so that structures inside register are properly treated.	`copy a`
filled	Checks whether a register has at least one item, including null. This is always true.	`filled a`
getentry	Returns entries without issuing an error if a given index does not exist.	`getentry(a, 1, 3)`
in	Checks whether an element is stored in the register, returns **true** or **false**.	`0 in reg(1, 0)`
pop bottom/ top	Pops the first or the last element from a register, shifting other elements to close the space, if necessary. Reduces the size of the register by one.	`pop bottom from a` `pop top from a`
size	Returns the number of `visible` elements.	`size a`
sort	Sorts a register in place. Please also see **sorted**.	`sort(a)`

Name	Description	Example
type	Returns the general type of a register, i.e. register.	`type a`
unique	Reduces multiple occurrences of an item in a register to just one.	`unique a`
unpack	Unpacks a register. See **unpack** in Chapter 8.	`unpack(a)`
duplicates	Finds duplicate elements.	`duplicates(a)`
map	Maps a function on all elements of a register.	`map(<< x -> x^2 >>, reg(1, 2, 3))`
purge	Removes the value at the given position and shifts all elements to close the space. Also reduces the size of the register by one.	
zip	Zips together two registers by applying a function to each of its respective elements.	`zip(<< x, y -> x + y >>, reg(1, 2), reg(3, 4))`
intersect	Searches all values in one register that are also values in another register and returns them in a new register.	`reg(1, 2) intersect reg(2, 3)`
minus	Searches all values in one register that are not values in another register and returns them as a new register.	`reg(1, 2) minus reg(2, 3)`
subset	Checks whether all values in a register are included in another register.	`reg(1) subset reg(1, 2)`
xsubset	Checks whether all values in a register are included in another register.	`reg(1) xsubset reg(1, 2)`
union	Concatenates two registers simply by copying all its elements.	`reg(1, 2) union reg(2, 3)`
setmeta-table	Assigns a metatable to a register.	`setmetatable (a, mtbl)`
getmeta-table	Returns the metatable stored to a register.	`getmetatable(a)`
registers. settop	Resets the top pointer to the given position, an integer.	
registers. reduce	Shrinks the size of a register to the given value.	
registers. extend	Enlarges the size of a register to the given value.	
environ. kernel/ regsize	Sets the default size of newly created registers the given value, a non-posint.	

Table 17: Some operators and functions applicable to registers

4.16 Exploring the Internals of Structures

If you would like to know how a table, set, sequence, register or pair is represented internally, please have a look at the **environ.attrib** function explained in Chapter 14.2. It might help when debugging code.

The function returns the estimated number of bytes used by a structure, how many slots have been pre-allocated and how many are actually occupied, whether a user-defined type has been set, how many elements have been allocated to the array and hash parts of a table, etc.

4.17 Other Types

For threads, userdata, and lightuserdata please refer to the Lua 5.1 documentation and Chapter 6.30.

Agena supports the following metamethods with userdata: =, ==, ~=, **size**, **in**, **union**, **intersect**, **minus**, **sumup**, and **qsumup**. `'__index'` , `'__writeindex'`, `'__gc'`, and `'__tostring'` are supported, as well.

Chapter **Five**

Control

5 Control

5.1 Conditions

Depending on a given condition, Agena can alternatively execute certain statements with either the **if** or **case** statement.
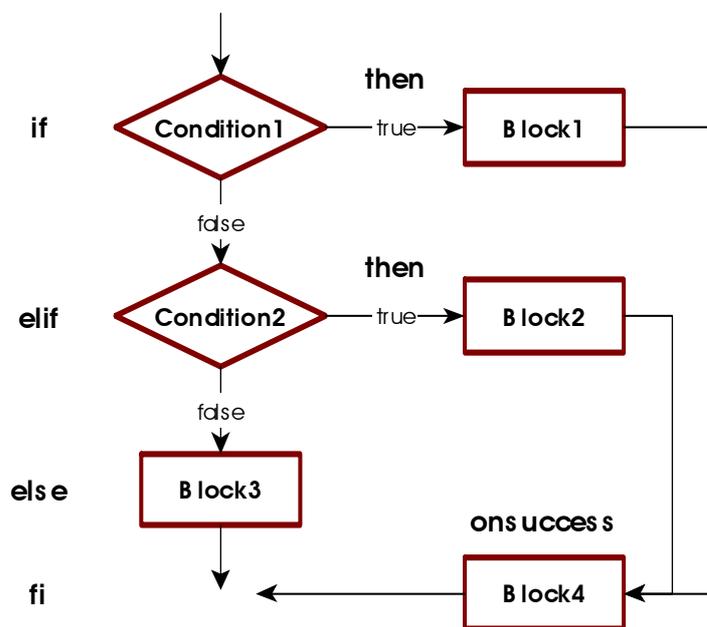
5.1.1 if Statement

The **if** statement checks a condition and selects one statement from many listed. Its syntax is as follows:

if *condition₁* **then**
    statements₁
[**elif** *condition₂* **then**
    statements₂]
[**onsuccess**
    statements₃]
[**else**
    statements₄]
fi

The condition may always evaluate to one of the Boolean values **true**, **false** or **fail**, or to any other value.

The **elif**, **else**, and **onsuccess** clauses are optional. While more than one **elif** clause can be given, only one **else** and one **onsuccess** clause is accepted.

If an **if** or **elif** condition results to **true** or any other value except **false**, **fail** or **null**, its corresponding **then** clause is executed. If all conditions result to **false**, **fail** or **null**, the **else** clause is executed if present - otherwise Agena proceeds with the next statement following the **fi** keyword.

If an **onsuccess** clause is given, and an **if** or **elif** condition results to **true**, the statements in this **onsuccess** branch are executed. This allows to move code common to all **then** clauses into one single branch, reducing code size. When

using both **onsuccess** and **else** clauses, the **onsuccess** clause must be put given before the else snippet.

Examples:

The condition **true** is always true, so the string 'yes' is printed.

```
> if true then
>    print('yes')
> fi;
yes
```

The next example demonstrates the behaviour if the condition is neither a Boolean nor **null**:

```
> if 1 then
>    print('One')
> fi;
One
```

In the following statement, the condition evaluates to **false**, so nothing is printed:

```
> if 1 <> 1 then
>    print('this will never be printed')
> fi;
```

An **if** statement with an **else** clause:

```
> if false then
>    print('this will never be printed')
> else
>    print('this will always be printed')
> fi;
this will always be printed
```

An **if** statement with an **elif** clause:

```
> if 1 = 2 then
>    print('this will never be printed')
> elif 1 < 2 then
>    print('this will always be printed')
> fi;
this will always be printed
```

An **if** statement with **elif** and **else** clauses:

```
> if 1 = 2 then
>    print('this will never be printed')
> elif 1 < 2 then
>    print('this will always be printed')
> else
>    print('neither will this be printed')
> fi;
this will always be printed
```

Sometimes certain conditions may just be skipped with an empty statement, denoted by **do nothing**, to make the code more readable:

```
> if 1 = 2 then
>    do nothing
> elif 1 < 2 then
>    print('this will always be printed')
> else
>    print('neither will this be printed')
> fi;
this will always be printed
```

One last example, this time demonstrating the optional **onsuccess** clause. As shown, both **then** statements include the same `flag := true` statement.

```
> if 1 = 2 then
>    print('this will never be printed');
>    flag := true
> elif 1 = 1 then
>    print('this will always be printed');
>    flag := true
> else
>    flag := false
> fi;
this will always be printed

> flag:
true
```

So the two assignment statements may be moved into one **onsuccess** clause.

```
> if 1 = 2 then
>    print('this will never be printed');
> elif 1 = 1 then
>    print('this will always be printed');
> onsuccess
>    flag := true
> else
>    flag := false
> fi;
this will always be printed

> flag:
true
```

if and **elif** statements also support simple assignments in the conditions, as well.

```
> if flag := true then
>    print('Output: ' & flag)
> fi;
Output: true
```

Only if the right-hand side of the assignment does neither result to **false**, **fail** nor **null**, will the corresponding **then** clause be executed.

You can also combine an assignment and a condition in the **if** clause:

```
> if c := 0, c >= 0 do
>    print(c)
> od;
0
```

5.1.2 if Operator, Version One

The **if** operator checks a condition and returns the respective expression.

[**with name₁, ⋯ := expr₁, ⋯ [->]**]
if *condition₁* **then** *expr₁* [**elif** *condition₂* **then** *expr₂*, ⋯] **else** *exprₖ* **fi**

The result is expression *expr₁* if *condition₁* is **true** or any other value except **false**, **fail** or **null**; and *exprₖ* otherwise. You can also optionally add one or more **elif** clauses.

Example:

```
> x := if 1 = 1 then true else false fi:
true
```

which is the same as:

```
> if 1 = 1 then
>    x := true
> else
>    x := false
> fi;
```

The **if** operator only evaluates the expression that it will return. Thus the other expression which will not be returned will never be checked for semantic correctness, e.g. out-of-range indices, etc. You may nest **if** operators.

An optional preceding **with** clause allows to define one or more auxiliary variables that are local to this operator only:

```
> x := Pi;

> a := with n := 2*x -> if x < 0 then n else 2*n fi;
```

which is syntactic sugar for:

```
> x := Pi;

> scope
>    local n := 2*x;
>    a := if x < 0 then n else 2*n fi
> epocs;
```

The arrow token is optional. Multiple auxiliary variables are defined as follows:

```
> a := with m, n := x, 2*x -> if x < 0 then m else n fi;
```

The **if** operator cannot return multiple values, only one.

5.1.3 if Operator, Version Two

There is a second operator form, reminiscent to the **if** statement; for example:

```
> a := 10;

> sgn := if is a < 0 then  # determines sign of `a'
>           print('I am negative');
>           [further statements ...]
>           return -1
>         elif a = 0 then
>           print('I am zero');
>           return 0
>         else
>           return 1
>         fi;
> sgn:
1
```

You may omit the **elif** and **else** clauses. Each clause may contain zero, one ore more statements, but it must always finish with the **return** expression which defines the resulting value (-1, 0 or 1 in the example above). In procedures, this special **return** expression does not cause a procedure to quit. Note that if the **else** clause is omitted, the operator returns **null** if no condition is met.

The operator returns exactly one value.

5.1.4 Short-cut Condition with ? and ?- Tokens

The question mark **?** expresses a short-cut `if`-like statement: if any condition preceding **?** evaluates to true, exactly one statement right behind the token is executed, otherwise the statement is simply skipped. Likewise, the **?-** token checks an expression and executes a one-line statement if it evaluates to **false**, **fail** or **null**.

```
> x := 0;

> x = 0 ? x := 1;

> x:
1


> x := 0;

> x <> 0 ?- x := 1;

> x:
1
```

5.1.5 case Statement

The **case** statement facilitates comparing values and executing corresponding statements. There exist two variants, the first one is:

```
case name
   [of value₁₁ [, value₁₂, · · ·] then statements₁
   [of value₂₁ to value₂₂ then statements₂]
   [of · · ·]
   [onsuccess · · ·]
   [else statementsₖ [esle]]
esac
```

```
> a := 'k';

> case a
>    of 'a', 'e', 'i', 'o', 'u', 'y' then
>        result := 'vowel'
>    else
>        result := 'consonant'
>    esle
> esac;

> result:
consonant
```

You can add as many **of/then** statements as you like. Fall through is not supported. This means that if one **then** clause is executed, Agena will not evaluate the following **of** clauses and will proceed with the statement right after the closing **esac** keyword. An **else** clause may be terminated by the **esle** token, but this is optional.

Instead of passing one or more individual values, you can also check whether a number x or the first character of a - non-empty - string x is part of a range a to b, i.e. $a \leq x \leq b$. One **to** range is accepted per **of** clause.

```
> a := 0;

> case a
>    of -1 then result := -1
>    of 0 to 10 then result := 10
>    of 'a' to 'c' then result := 0
> esac;
```

As with the **if** statement, if an **onsuccess** clause is given, and in case one of the conditions results to **true**, the statements in the **onsuccess** branch are executed. This allows to move code common to all **then** clauses into one single branch, reducing the code size.

If none of the **of** conditions is satisfied, and if an **else** clause is given, then the respective **else** statements are processed, otherwise Agena executes the code following the **esac** token.
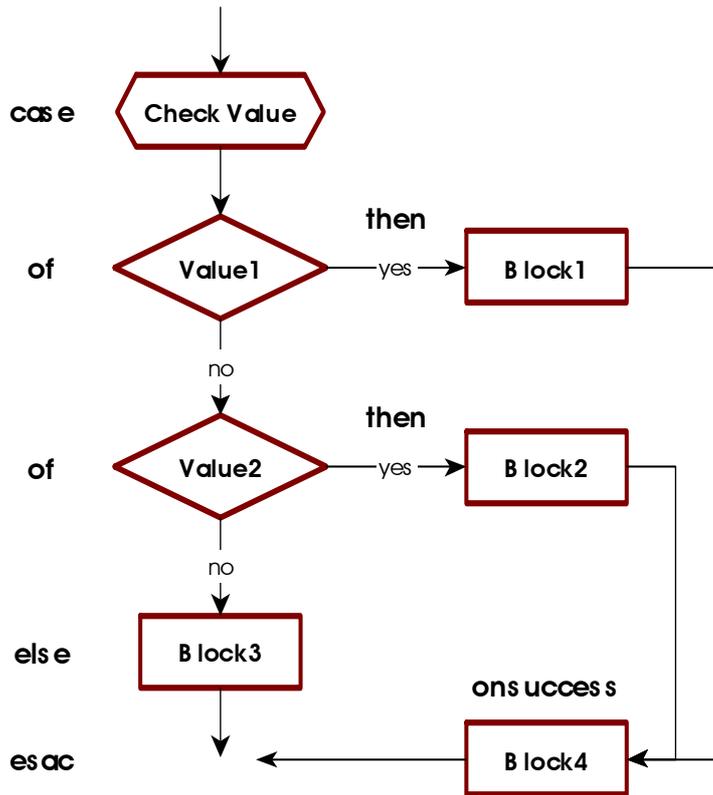
The second variant is exactly equal to the **if** statement but may improve the readability of programme code.

With both variants, instead of the **then** keyword the `->` token can be used.

5.1.6 case of Statement

A flavour of the **if** statement is the **case of** control. It may improve the readability of code.

There is no functional difference between **if** and **case of** statements.

Example:

```
> x := 0; flag := false;

> case
>    of x < 0 then r := -1;
>    of x = 0 then r := 0;
>    onsuccess flag := true;
>    else r := 1 esle
> esac

> r, flag:
0    true
```

```
case
  of condition₁ then statements₁
  [of condition₂ then statements₂]
  [of · · ·]
  [onsuccess · · ·]
  [else statementsₖ [esle]]
esac
```

case of statements also support simple assignments in the **case of** clause, and their optional **of** clauses, as well.
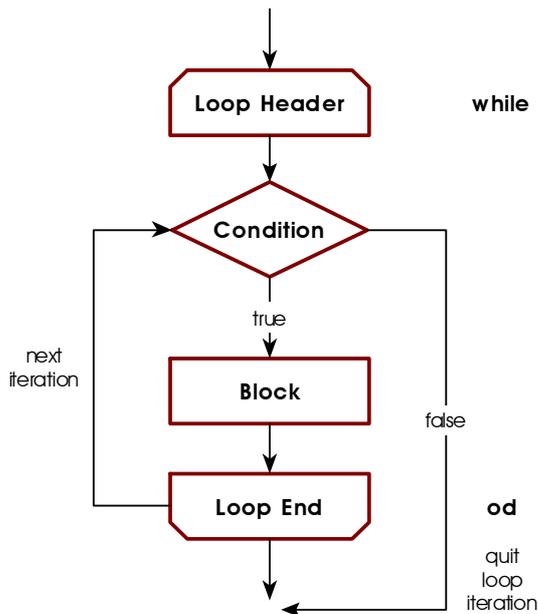
```
> case of flag := io.read() then
>    print('Output: ' & flag)
> esac;
Agena
Output: Agena
```

Only if the right-hand side of the assignment does neither result to **false**, **fail** nor **null** will the **then** clause be executed.

5.2 Loops

Agena has three basic forms of control-flow statements that perform looping: **while** and **for**, each with different variations.

5.2.1 while Loops

A **while** loop first checks a condition and if this condition is **true** or any other value except **false**, **fail** or **null**, it iterates the loop body again and again as long as the condition remains true.

If the condition is **false**, **fail** or **null**, no further iteration is done and control returns to the statement following right after the loop body.

If the condition is **false**, **fail** or **null** right from the start, the loop is not executed at all.

> **while** *condition* **do**
>   *statements*
> **od**

The programme flow is as shown in the diagram above.

The following statements calculate the largest Fibonacci number less than 1000.

```
> a := 0; b := 1;

> while b < 1000 do
>    c := b;
>    b := a + b;
>    a := c
> od;

> c:
987
```

The following loop will never be executed since the condition is **false**:

```
> while false do
>    print('never printed')
> od;
```

You can also conduct a simple assignment in the **while** condition. If an assignment is given in the **while** clause, its right-hand side is evaluated and stored to the left-hand side name. The result of the evaluation is then checked and either the loop body is executed - the result of the evaluation is neither **false**, **fail** nor **null** - or not.

This allows for shorter code: Instead of

```
> flag := true;
> while flag do
>    flag := io.read();
>    if flag = 'Z' then break fi
> od
```

you can now simply write (no need to assign `flag` before):

```
> while flag := io.read() do
>    if flag = 'Z' then break fi
> od
```

The variable assigned in the **while** clause is *not* local to the loop body but can be accessed later on the level that surrounds the loop. You may explicitly declare the variable local before.

You can also combine an assignment and a condition in the **while** clause. In this case, the assignment will be done only once and the condition will always be checked:

```
> while c := 0, c < 3 do
>    print(c++)
> od;
0
1
2

> c:
3
```

Variations of **while** are the **do/as** and **do/until** loops which check a condition at the end of the iteration, and thus will always be executed at least once.

In the **do/as** variant, as long as the condition evaluates to **true**, the loop body is executed.

```
> c := 0;

> do
>    inc c
> as c < 10;


> c:
10
```

```
do
    statements
as condition
```

do/**until** loops are iterated until the given condition is met.

```
> c := 0;

> do
>    inc c
> until c > 10;

> c:
11
```

```
do
    statements
until condition
```

do/**as** and **do**/**until** support simple assignments in the respective condition.

Another flavour of the **while** loop is the infinite **do**/**od** loop which executes statements infinitely and can be interrupted with the **break** or **return** statements. See Chapter 5.2.10 for further information on the **break** statement. It is syntactic sugar for the **while true do**/**od** construct.

```
do
    statements
od
```

```
> i := 0;

> do
>    inc i;
>    if i > 3 then break fi;
>    print(i)
> od;
1
2
3
```
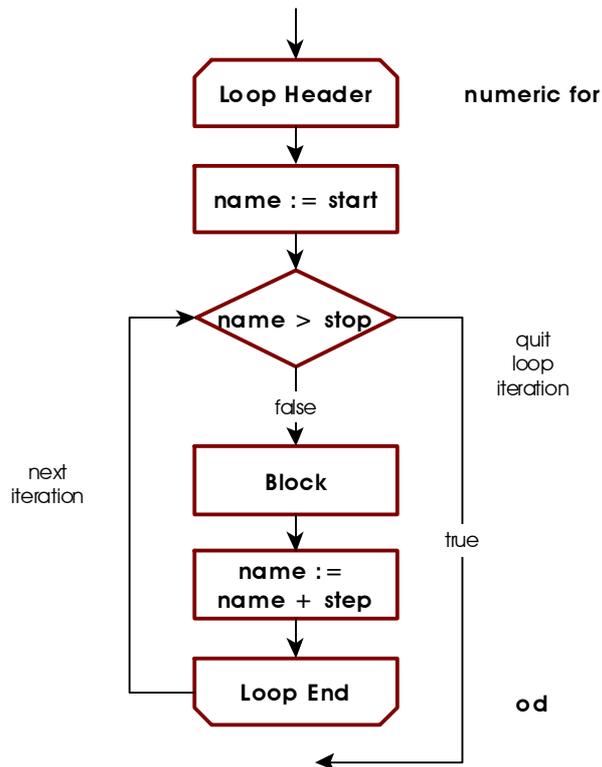
for loops are used if the number of iterations is known in advance. There are **for**/**to** loops for numeric progressions, and **for**/**in** loops for table and string iterations.

5.2.2 for/to Loops

Let us first consider numeric **for/to** loops which use numeric values for control:

> **for** *name* [**from** *start*] [**to** *stop*]
>   [**by** *step*] **do**
>   *statements*
> **od**

name, *start*, *stop*, and *step* are all numeric values or must evaluate to numeric values.

The statement at first sets the variable *name* to the value of *start*. *name* is called the *control* or *loop variable*. If *start* is not given, the start value by default is $+1$.

When omitting the **to** clause, the loop iterates until the largest number representable on your platform has been reached. If left out, the *step* size is $+1$.

The **for** loop then checks whether *start* $\leq$ *stop*. If so, it executes *statements* and returns to the top of the loop, increments *name* by *step* and then checks whether the new value is less or equal *stop*. If so, *statements* are executed again.

```
> for i from 1 to 3 by 1 do
>    print(i, i^2, i^3)
> od;
1        1        1
2        4        8
3        9        27

> for i to 3 do
>    print(i, i^2, i^3)
> od;
1        1        1
2        4        8
3        9        27
```

The control variable of a loop is always accessible to its surrounding block, so you may use its value in subsequent statements. This rule applies only to **for/from/to**-loops with or without a **while**, **as** or **until** extension, but not to for/in loops

described below. Note that within procedures, the loop control variable is automatically declared local, while on the interactive level it is global.

```
> for i while fact(i) < 1k do od

> i:
7
```

The following rules apply to the value of the control variable after leaving the loop:

1. If the loop terminates normally, i.e. if it iterates until the stop value has been reached, then the value of the control variable is its stop value *plus* the step size.

2. If the loop is left prematurely by executing a **break** statement[13] within the loop, or if a **for/while** loop is terminated because the **while** condition evaluated to **false** (see Chapter 5.2.8), then the control variable is set to the loop's last iteration value before quitting the loop. There will be no increment with the loop's step size. The same applies to **for/as** and **for/until** loops (see Chapter 5.2.9).

Loops can count backwards if the step size is negative (see also the next chapter):

```
> for i from 2 to 1 by -1 do
>    print(i)
> od
2
1
```

A special form is the **to/do** loop which does not feature a control variable and iterates exactly n times.

```
> to 2 do
>    print('iterating')
> od
iterating
iterating
```

Agena automatically uses an advanced precision algorithm based on Neumaier summation if the step size is non-integral, e.g. 0.1, -0.01. This mostly prevents round-off errors, thus avoids that the loop stops before the last iteration value - the limit - has been reached and that iteration values with round-off errors are returned. You may switch Agena into Kahan-Ozawa or Kahan-Babuška summation mode to use extended round-off prevention by issuing the statement in a session:

```
> environ.kernel(kahanozawa = true);
```

or

```
> environ.kernel(kahanbabuska = true);
```

---

[13] See Chapter 5.2.8 for more information in the **break** statement.

As a further measure to prevent a loop stopping before the stop limit has been reached, numeric **for** loops with fractional step sizes automatically increase the stop limit by the value of the constant **hEps**. If a you pass a step size that is equals or less then **hEps**, Agena now issues an error. You can entirely switch off this **math.Eps** to zero, but only by calling **environ.kernel**:

```
> environ.kernel(hEps = 0);
```

Kahan-Babuška summation may be more accurate than Kahan-Ozawa summation. The speed loss with both algorithms compared to Neumaier is around 20 percent or more.

If the step size is an integer, e.g. 1000, 1, -1.0, then Agena does not use advanced precision to ensure maximum speed.

5.2.3 for/downto Loops

count from a **start** value *down* to a **stop** value, with a default countdown **step** size of (implicit minus) one. To count down, the optional **step** size should be positive.

> **for** *name* **from** *start* **downto** *stop* [**by** *step*] **do**
>   *statements*
> **od**

5.2.4 for/in Loops over Tables

are used to traverse tables, strings, sets, and sequences, and also iterate over functions.

If **null** is passed after the **in** keyword, or if the value evaluates to **null**, then Agena does not execute the loop and continues with the statement following it.

Let us first concentrate on table iteration.

> **for** *key*, *value* **in** *tbl* **do**
>   *statements*
> **od**

The loop iterates over all key~value pairs in table *tbl* and with each iteration assigns the respective key to *key*, and its value to *value*.

```
> a := [4, 5, 6]
> for i, j in a do
>    print(i, j)
> od
1       4
2       5
3       6
```

There are two variations: When putting the token **keys** in front of the control variable, the loop iterates only over the keys of a table:

> **for keys** *key* **in** *tbl* **do**
>   *statements*
> **od**

Example:

```
> for keys i in a do
>    print(i)
> od
1
2
3
```

The other variation iterates on the values of a table only:

> **for** *value* **in** *tbl* **do**
>   *statements*
> **od**

```
> for i in a do
>    print(i)
> od
4
5
6
```

The control variables in **for/in** loops are always local to the body of the loop (as opposed to numeric **for** loops). You may assign their values to other variables if you need them later.

You should never change the value of the control variables in the body of a loop - the result would be undefined. Use the **copy** function to safely traverse any structure if you want to change, add, or delete its entries.

Because of the implementation of tables, please note that the keys in a table are not necessarily traversed in ascending order. You may want to iterate sequences or

linked lists (see Chapter 6.27).

5.2.5 for/in Loops over Sequences and Registers

All of the features explained in the last subchapter are applicable to sequences and registers, as well.

5.2.6 for/in Loops over Strings

If you want to iterate over a string character by character from its left to its right, you may use a **for/in** loop as well. All of the variations are supported.

> **for** *key*, *value* **in** *string* **do** *statements* **od**
>
> **for** *value* **in** *string* **do** *statements* **od**
>
> **for keys** *value* **in** *string* **do** *statements* **od**

The following code converts a word to a sequence of abstract vowel, ligature, and consonant place holders and also counts their respective occurrence:

```
> str := 'æfter';

> result := '';

> c, v, l -> 0;

> for i in str do
>    case i
>       of 'a', 'e', 'i', 'o', 'u' then
>          result &:= 'V';
>          inc v
>       of 'å', 'æ', 'ø', 'ö' then
>          result &:= 'L';
>          inc l
>    else
>          result &:= 'C'
>          inc c
>    esac
> od;

> print(result, v & ' vowels', l & ' ligatures', c & ' consonants');
LCCVC       1 vowels         1 ligatures        3 consonants
```

5.2.7 for/in Loops over Sets

All **for** loop variations support sets, as well. The only useful one, however, is the following:

```
> sister := {'swistar', 'sweastor', 'svasar', 'sister'}

> for i in sister do print(i) od;
```

```
svasar
swistar
sweastor
sister
```

You may try the other loop alternatives to see what happens.

5.2.8 for/in Loops over Procedures

The following procedure, called an iterator, returns a sequence of values multiplied by two. If $state = n$, then the procedure returns **null**, quitting the **for/in** iteration. Note that the iterator in its first result n returns the next value of the loop control variable i. We use $state$ to hold the number of iterations we wish to perform. See Chapter 6 which describes procedures in detail.

```
> double := proc(state, n) is
>    if state > n then
>        inc n;
>        return n, 2*n
>    else
>        return null
>    fi
> end;
```

In the following loop, 5 denotes the state and 0 the initial value.

```
> for i, j in double, 5, 0 do
>    print(i, j)
> od
1        2
2        4
3        6
4        8
5        10
```

Another means to iterate over procedures are closures, see Chapter 6.22. So far, here is just an example that you can use as a template for further experiments:

```
> iterate := proc(obj) is
>    local n := 0;        # with each call, counts up by one
>    return proc() is
>        inc n;
>        if n <= size obj then
>           return n, obj[n]
>        else
>           return null  # quit iteration
>        fi
>    end
> end;

> f := iterate(seq(Pi, 2*Pi, 3*Pi));

> for i, j in f do
>    print(i, j)
> od;
```

```
1          3.1415926535898
2          6.2831853071796
3          9.4247779607694
```

You might also use the generic **ipairs** and **pairs** functions with **for**/**in** loops:

ipairs iterates table arrays, sequences, registers, strings and userdata that have an '__index' metamethod, in a standard way:

```
>  for i, j in ipairs(s) do
>     print(i, j)
>  od;
1          3.1415926535898
2          6.2831853071796
3          9.4247779607694

> import numarray

> d := numarray.double(3)

> for i to 3 do d[i] := i*Pi od

> for i, j in ipairs(d) do
>    print(i, j)
> od
1          3.1415926535898
2          6.2831853071796
3          9.4247779607694
```

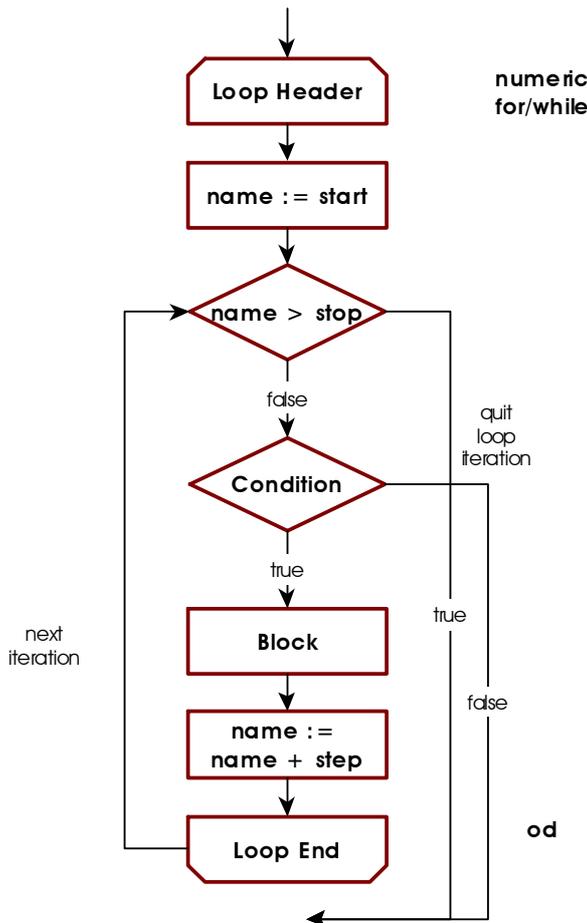To check whether a userdata features an '__index' entry in its associated metatable, just enter:

```
> getmetatable(d).__index:
procedure(01CE6DD0)
```

pairs allows to iterate all the keys and corresponding values of a dictionary, but as the following example shows, not surprisingly in a `random` fashion:

```
> t := [a = Pi, b = 2*Pi, c = 3*Pi]

> for i, j in pairs(t) do
>    print(i, j)
> od
a          3.1415926535898
c          9.4247779607694
b          6.2831853071796
```

Take in mind that **ipairs** and **pairs** are much slower than iterating structures directly.

**numeric
for/while**

Loop Header

name := start

name > stop

false

*quit
loop
iteration*

Condition

true

true

*next
iteration*

Block

false

**name :=
name + step**

Loop End

od

5.2.9 for/while and for/until Loops

All flavours of **for** loops can be combined with a **while** condition. As long as the **while** condition is satisfied, the **for** loop iterates. To be more precise, before Agena starts the first iteration of a loop or continues with the next iteration, it checks the while condition to be **true** or any other value except **false**, **fail** or **null**. An example:

```
> for x to 10
>    while ln(x) <= 1 do
>       print(x, ln(x))
> od
1       0
2       0.69314718055995
```

Regardless of the value of the **while** condition, the loop control variables are always initiated with the start values: in the summary frame below, with **for/to** loops, *a* is assigned to *i* (or 1 if the **from** clause is not given); *key* and/or *value* are assigned with the first

item in the table, set or sequence *struct* or the first character in string *string*. Likewise, the **until** condition quits a loop until it is satisfied.

> **for** *i* [**from** *a*] [**to** *b*] [**by** *step*] (**while**|**until**) *condition* **do** *statements* **od**
> **for** [*key*,] *value* **in** *struct* (**while**|**until**) *condition* **do** *statements* **od**
> **for keys** *key* **in** *struct* (**while**|**until**) *condition* **do** *statements* **od**
> **for** [*key*,] *value* **in** *string* (**while**|**until**) *condition* **do** *statements* **od**
> **for keys** *key* **in** *string* (**while**|**until**) *condition* **do** *statements* **od**

The optional **while** and **until** clauses accept a simple assignment. In such a case, the right-hand side of the assignment is evaluated and stored to the left-hand side non-local name. The result of the evaluation is then checked and either the loop body is executed or not. Example:

```
> a := [10, 20, 4 ~ 30]  # the table has no index 3

> for i to 4 while t := a[i] do  # since a[3] evaluates to null,
>    # which is equal to false in this context, the loop quits with i = 3.
>    print(a[i], t)
> od
10          10
20          20
```

5.2.10 for/as & for/until Loops

As with the optional **while** clause, all flavours of **for** loops can be combined with an **as** or an **until** condition.

In these cases, a loop is always iterated at least once, and after the first iteration is completed, Agena checks the given condition and decides whether to start the next iteration or to leave the loop.

In the following example, the **for/as** loop starts with $i=0$ and since the first check to the **as** condition results to **true**, the next iteration with $i=1$ is conducted. The next check to the **as** condition results to **false**, thus the loop quits.

```
> for i from 0 do
>    print(i, 10^i)
> as 10^i < 10
0        1
1        10
```

The next loop iterates three times, until $i=2$, since only then the **until** condition becomes **true**.

```
> for i from 0 do
>    print(i, 10^i)
> until 10^i > 10
0        1
1        10
2        100
```
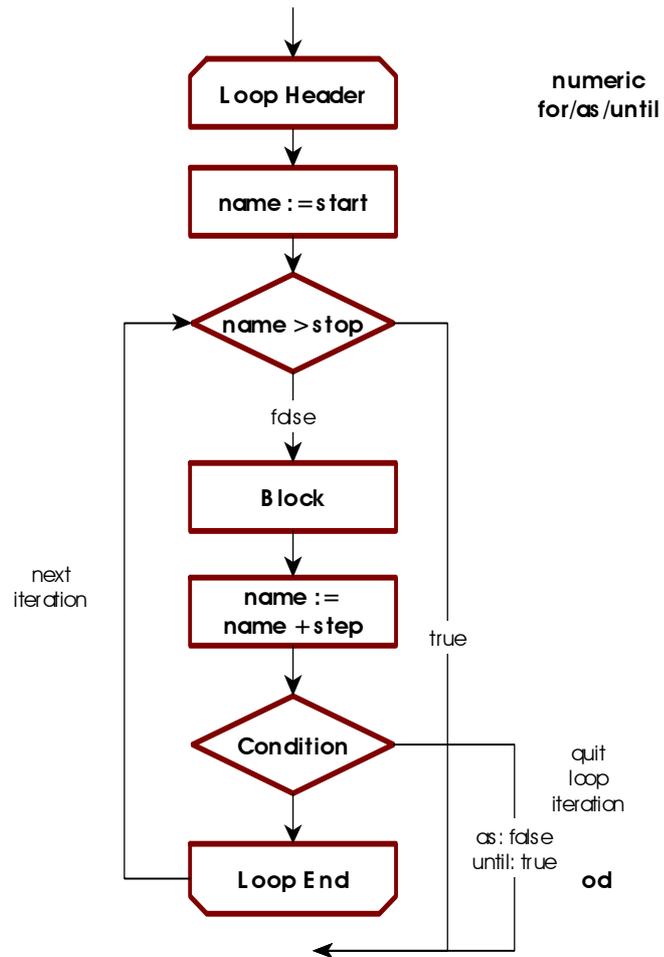
5.2.11 Loop Jump Control

Agena features statements to manipulate loop execution. **skip** and **break** are applicable to all loop types, whereas **redo** and **relaunch** work in **for** loops only.

The **skip** statement causes another iteration of the loop to begin at once, thus skipping all of the loop statements following it.

The **break** statement quits the execution of the loop entirely and proceeds with the next statement right after the end of the loop.
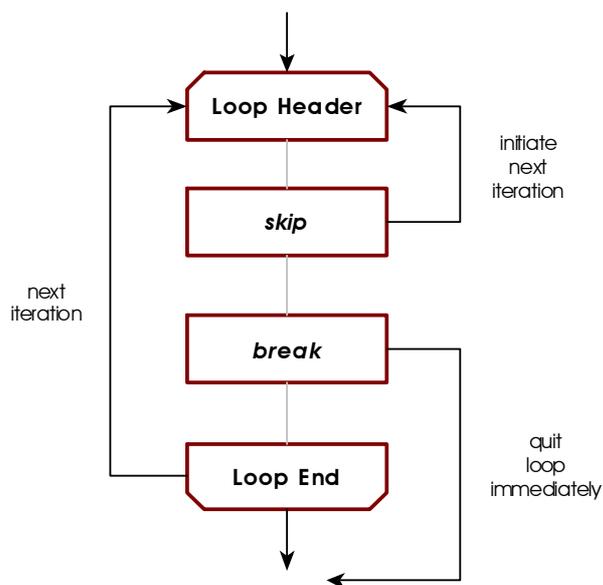

```
> for i to 5 do
>    if i = 3 then skip fi;
>    print(i)
>    if i = 4 then break fi;
> od;
1
2
4
```

This is equivalent to the following statement:

```
> for i to 5 while i < 5 do
>    if i = 3 then skip fi;
>    print(i)
> od;
1
2
4
```

```
> a := 0;

> while true do
>    inc a;
>    if a > 5 then break fi;
>    if a < 3 then skip fi;
>    print(a)
> od;
3
4
5
```

There exists syntactical sugar for both the **skip** and the **break** statements: instead of putting these statements into **if** clauses, just add the **when** token along with a condition to the respective keyword.

```
> a := 0;

> while true do
>    inc a;
>    break when a > 5;
>    skip when a < 3;
>    print(a)
> od;
```

```
3
4
5
```

In **for/to** and **for/in** loops, the **redo** statement is similar to **skip**: it jumps back to the beginning of the loop but does not change the loop control variable in **for/to** loops or the index/value control variables in **for/in** loops. Thus, it restarts the *current* iteration. At restart, it checks an optional **while** condition, if present.

```
> flag := true;

> for j in [10, 11, 12] do
>    print(j, flag);
>    if flag and j = 11 then
>       clear flag;
>       print(j, flag,
>          'jump back')
>       redo
>    fi;
> until j > 12;

10    true
11    true
11    false     jump back
11    false
12    false
```
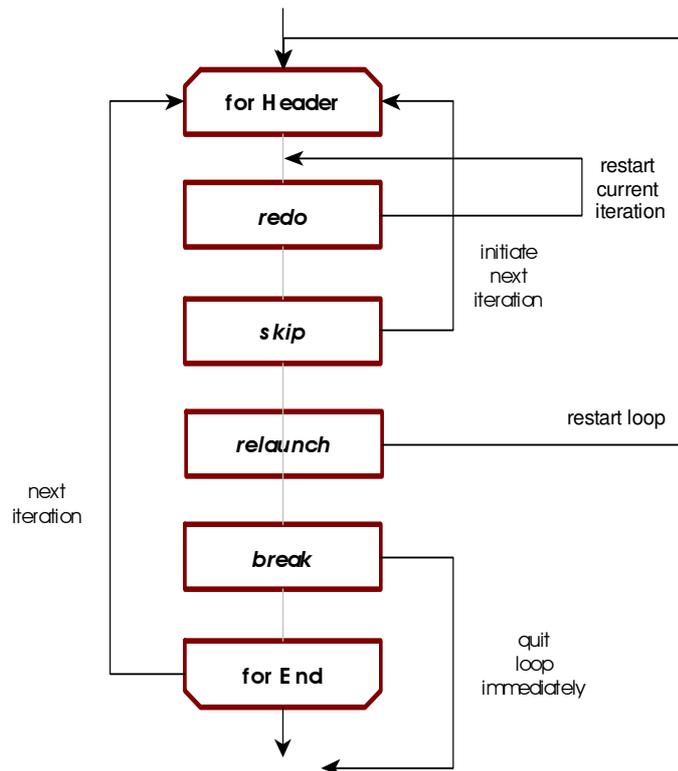
The **relaunch** statement completely restarts a **for/to** and **for/in** loop from its very beginning, i.e. resets the current control variable to its start value (**from** clause or first element, respectively).

```
> flag := true;

> for j in [10, 11, 12] do
>    print(j, flag);
>    if flag and j = 11 then
>       clear flag;
>       print(j, flag,
>          'restart')
>       relaunch
>    fi;
> until j > 12;
10    true
11    true
11    null    restart
10    null
11    null
12    null
```

5.2.12 Scope I: scope and epocs

You can define the scope of local variables with the **scope**/**epocs** statement. Any variable declared local between the **scope** and **epocs** keywords exists only in this block, and they are not available outside of it:

> **scope**
>   *declarations and statements*
> **epocs**

An example:

```
> a := 2;

> scope
>    local b := 3;  # b is local to the scope only
>    c := a*b       # c is available outside the block
> epocs;

> print(a, b, c);
2       null   6
```

5.2.13 Scope II: with Statement

The **with** statement allows to define a scope and assign one or more local variables in only one stroke. It is syntactic sugar to the scope statement only. The following example refers to the example in the preceding subchapter:

> **with** $name_1$, $\cdots$:= $expr_1$, $\cdots$ **do**
>   *declarations and statements*
> **od**

```
> a := 2;

> with b := 3 do  # b is local, a and c are global
>    c := a*b
> od;

> print(a, b, c);
2        null   6
```

Assign multiple local variables, in this case two variables:

```
> a := 2;

> with b, c := 3, 4 do
>    d := a*b*c
> od;

> print(a, b, c, d);
2        null   6       24
```

5.2.14 with Statement for Dictionaries

The **with** statement can also unpack table values, indexed by string keys, declare them local and then access them in the respective block. After leaving the block, all the values listed right between the **with** and **in** tokens are automatically written back to the table:

> **with** *key₁ [, key₂,, ···]* **in** *tablename* **do**
>   *statements*
> **od**

```
> zips := ['duedo' ~ 40210:40629,
>          bonn    = 53111:53229,
>          cologne = 50667:51149];

> with duedo, cologne in zips do  # bonn has not been given here
>    print(duedo, bonn, cologne);
>    cologne := null;  # cologne entry will be deleted from table zips
>    duedo := 40210:51149 # duedo entry in zips will be changed
>    # bonn entry will not be changed since not listed in the header
>    bonn := null
>    print(bonn, cologne, duedo)
> od;
40210:40629     null    50667:51149
null    null    40210:51149

> zips:
[bonn ~ 53111:53229, duedo ~ 40210:51149]
```

Another flavour of the **with** statement has the following syntax:

> **with** *tablename* **do**
>   *statements*
> **od**

Within the body of this variant, the table *tablename* can be referenced by just an underscore. It also allows to actively change values in *tablename*. Example:

```
> zips := [duedo = 4000, bonn = 5300]

> with zips do
>    print(_.bonn);
>    _.bonn := 53111
> od
5300

> zips:
[bonn ~ 53111, duedo ~ 4000]
```

5.2.15 Alternative to Closing Keywords

You can use the **end** token instead of the closing **fi**, **od**, **esac**, **yrt** and **epocs** keywords, or mix both.

Example:

```
> if os.system()[1] in {'SunOS', 'Windows', 'Linux', 'Darwin'} then
>    if environ.kernel().is32bit then
>       readlib('fractals');
>       readlib('gdi');
>       readlib('gzip');
>       a, b := gzip.deflate('agena programming language');
>       if [gzip.inflate(a, b)] <> ['agena programming language', 26] then
>          print('error in gzip.in\\deflate')
>       end;
>       try  # provoke segfaults
>         for i from 0 to 100 do
>            gzip.inflate(a, i)
>         od
>       end
>    end;
>    to 100 do readlib('net') end  # try crashing Agena at exit
> fi;
```

Chapter **Six**

Programming

6 Programming

Writing effective code in a minimum amount of time is one of the key features of Agena. Programmes are usually represented by procedures. The words `procedure` and `function` are used synonymously in this text.

6.1 Procedures

In general, procedures conflate a sequence of statements into abstract units which then can be repeatedly invoked.

Writing procedures in Agena is quite simple:

> *procname* := **proc**([*par₁* [**::***type₁*] [, *par₂* [**::***type₂*], ···]]) [**::** *returntype*] **is**
>   [**local** *name₁* [, *name₂*,···]];
>   *statements*
> **end**

All the values that a procedure shall process are given as *parameters par₁*, etc. A function may have no, one or more parameters. A parameter may be succeeded by the name of a type (see Chapter 6.8.2), or a set of up to four types, that an argument must satisfy when the procedure is called.

If a type is given right after the parameter list, Agena checks whether the return of the procedure is of the given *returntype*, which may also be a user-defined type. The **is** keyword is obligatory.

A procedure usually uses local variables which are private to the procedure and cannot be accessed by other procedures or on the Agena interactive level.

Global variables are supported in Agena, as well. All values assigned on the interactive level are global, and you can also create global variables within a procedure. The values of global variables can be accessed on the interactive level and within any procedure.

A procedure may call other functions or itself. A procedure may even include definitions of further local or global procedures.

The result of a procedure will be returned through the **return** keyword which may be put anywhere in the procedure body, and which also immediately terminates execution of the procedure.

> **return** [*value* [, *value₂*,···]]

As you can see, you may not only return a single result, but also multiple ones, or none at all.

Furthermore, a procedure will not return anything - not even the **null** value -

- if no **return** statement is given at all,
- if no values are given in the **return** statement.

The following procedure computes the factorial of an integer[14]:

```
> restart;

> fact := proc(n) is
>     # computes the factorial of an integer n
>     if n < 0 then return fail
>     elif n = 0 then return 1
>     else return fact(n-1)*n
>     fi
> end;
```

It is invoked using the syntax:

$$\textit{funcname}(\textbf{[}\textit{arg}_1 \textbf{ [,} \textit{arg}_2, \cdots \textbf{]])}$$

```
> fact(4):
24
```

where the first parameter is replaced by the first argument arg_1, the second parameter is substituted with arg_2, etc.

When calling a function recursively, instead of writing out its real name, you may use the **procname** keyword, which in runtime is substituted by the name with which the procedure was invoked:

```
> fact := proc(n) is
>     # computes the factorial of an integer n
>     if n < 0 then return fail
>     elif n = 0 then return 1
>     else return procname(n-1)*n
>     fi
> end;
```

A **when** clause can be added to a **return** statement that does not pass back any value including **null**. In this case, the execution of a function is being finished if the Boolean **when** condition has been satisfied, e.g. `return when x <> 0`. **return** can be combined with both a **when** and **with** clause - for example

```
> return when x <> 0 with true;
```

is syntactic sugar for

```
> if x <> 0 then
>     return true
> fi;
```

---

[14]The library function **fact** is much faster.

Last of all, procedures can alternatively be defined as follows:

> [**local**] **proc** *procname*([*par₁* [*::type₁*] [, *par₂* [*::type₂*], ···]]) [**::** *returntype*] **is**
>   [**local** [**constant**] *name₁* [, [**constant**] *name₂*, ···]];
>   *statements*
> **end**

Instead of the **proc** keyword, you can use the **procname** token. Thus, the factorial function can also be entered as follows:

```
> proc fact(n) is
>    if n < 0 then return fail
>    elif n = 0 then return 1
>    else return procname(n-1)*n
>    fi
> end;
```

6.2 Local Variables

The function above does not need local variables as it calls itself recursively. However, with large values for n, the large number of unevaluated recursive function calls will ultimately cause stack overflows. So we should use an iterative algorithm to compute the factorial and store intermediate results in a local variable.

A local variable is known only to the respective procedure and the block where it has been declared. It cannot be used in other procedures, the interactive Agena level, or outside the block where the local variable has been declared.

A local variable can be declared explicitly anywhere in the procedure body, but at least before its first usage. If you do not declare a variable as local and assign values later to this variable, then it is global. Note that control variables in **for** loops are always implicitly declared local to either their surrounding (**for/to** loops) or inner block (**for/in** loops), so we do not need to explicitly declare them.

Local declarations come in different flavours:

> **local** *name₁* [, *name₂*, ···]
> **local** [**constant**] *name₁* [, [**constant**] *name₂*, ···] **:=** *value₁* [, *value₂*, ···]
> **local** [**constant**] *name₁* [, [**constant**] *name₂*, ···] **->** *value*
> **local enum** *name₁* [, *name₂*, ···] [**from** *value*]
> **local** *key₁* [, *key₂*, ···] **in** *tablename*

In the first form, *name₁*, etc. are declared local.

In the second and third form, *name₁*, etc. are declared local and, as opposed to the first form, followed by initial assignments of values to these names.

In the fourth form, *name₁*, etc. are declared local and subsequently enumerated, i.e. assigned integers in ascending order, by default starting from 1, or the integer given in the optional **from** clause.

In the last form, table values are unpacked, equivalent to the assignment statement *key₁*, *key₂*, etc. := *tablename.key₁*, *tablename.key₂*, etc., with *key₁*, *key₂*, etc. being automatically declared local.

By passing the **constant** keyword in front of a variable name, a variable will become a constant that cannot be changed later in a session. This feature works in procedures only, not on the interactive level.

Let us write a procedure to compute the factorial using a **for** loop. To avoid unnecessary loop iterations when the intermediate result has become so large that it cannot be represented as a finite number, we also add a clause to quit loop iteration in such a case.

```
> fact := proc(n) is
>    if n < 0 then return fail fi;
>    local result := 1;
>    for i from 1 to n do
>       result := result * i
>       if not finite(result) then break fi
>    od;
>    return result
> end;

> fact(10):
3628800
```

Since `result` has been declared local it does not exist on the interactive level:

```
> result:
null
```

There is a shortcut to create local structures - tables, sets, and sequences:

> **create local** <structure> *name₁* [, <structure> *name₂*, · · ·]

where <structure> might be the keyword **table**, **set** or **sequence**. You can declare different local structures with one **create local** statement.

A useful function is **environ.globals** which determines global variable assignments inside procedures and helps to find those positions where a local declaration has been forgotten.

6.3 Global Variables

Global variables are visible to all procedures and the interactive level, such that their values can be queried and altered everywhere in your code.

Using global variables is not recommended. However, they are quite useful in order to have more control on the behaviour of procedures. For example, you may want to define a global variable `_EnvMoreInfo` that is checked in your procedures in order to print or not to print information to the user.

Global variables can be depicted with the **global** statement. It checks whether the given variable or variables have not been declared local before its execution and issues an error otherwise.

```
> fact := proc(n) is
>    if n < 0 then return fail fi;
>    local result := 1;
>    global _EnvMoreInfo;
>    for i from 1 to n do
>       result := result * i
>       if result = infinity then
>          if _EnvMoreInfo then print('Overflow !') fi;
>          break
>       fi
>    od;
>    return result
> end;
```

We should assign `_EnvMoreInfo` any value different from **null**, **fail** or **false** in order to get a warning message at runtime.

```
> _EnvMoreInfo := true;

> fact(10000):
Overflow !
infinity
```

6.4 Changing Parameter Values

You can change the values of procedure parameters within a procedure. Thus, an alternative to the **abs** operator might be:

```
> myAbs := proc(x) is
>    if x < 0 then
>       x := -x
>    fi;
>    return x
> end;

> myAbs(-1):
1
```

6.5 Optional Arguments

A function does not have to be called with exactly the number of parameters given at procedure definition. You may also pass less or more values. If no value is passed for a parameter, then it is automatically set to **null** at function invocation. If you pass more arguments than there are actual parameters, excess arguments are ignored.

For example, we can control whether a warning message is printed during function execution by passing an optional argument:

```
> fact := proc(n, warning) is
>    if n < 0 then return fail fi;
>    local result := 1;
>    for i from 1 to n do
>       result := result * i
>       if result = infinity then
>          if warning then print('Overflow !') fi;
>          break
>       fi
>    od;
>    return result
> end;

> fact(10000):
infinity
```

In this example, the option must be any value other than **null**, **false** or **fail** to get the effect.

```
> fact(10000, true):
Overflow !
infinity
```

A variable number of arguments can be passed by indicating them with a question mark in the parameter list and then querying them with the **varargs** system table in the procedure body. The **?** token can be used within in the procedure body as a shortcut to the **varargs** table.

```
> varadd := proc(?) is
>    local result := 0;
>    for i to size ? do
>       inc result, ?[i]
>    od;
>    return result
> end;

> varadd(1, 2, 3, 4, 5):
15
```

You may determine the number of arguments *actually* passed in a procedure call by querying the system variable **nargs** inside the respective procedure. A variant of the above procedure might thus be:

```
> varadd := proc(?) is
>    local result := 0;
>    for i to nargs do
>       inc result, ?[i]
>    od;
>    return result
> end;

> varadd(1, 2, 3, 4, 5):
15
```

Note: With OOP-style methods, **nargs** will also count the method itself.

Let us build an extended square root function that either computes in the real or complex domain. By default, i.e. if only one argument is given, the real domain is taken, otherwise you may explicitly set the domain using a pair as a second argument.

```
> xsqrt := proc(x, mode) is
>    if nargs = 1 or mode = 'domain':'real' then
>       return sqrt(x)
>    elif mode = 'domain':'complex' then
>       return sqrt(x + 0*I)
>    else
>       return fail
>    fi
> end;

> xsqrt(-2):
undefined

> xsqrt(-2, 'domain':'real'):
undefined
```

If the left-hand value of the pair in a function call shall denote a string, you can spare the single quotes around the string by using the = token which converts the left-hand name to a string[15].

```
> xsqrt(-2, domain = 'complex'):
1.4142135623731*I
```

You can mix optional arguments and the variable-arguments feature in parameter lists, with the question mark always the last item in the list:

```
> xsqrt := proc(x, mode, ?) is
>    ...
> end;
```

Finally, if you would like to define defaults for missing arguments, just use the binary **or** operator as shown below as it returns the first operand if it is non-**null**, and it returns the second operand if the first is **null**:

```
> f := proc(x) is
>    x := x or 0;
>    return x
> end;

> f():
0
```

---

[15] If you need a Boolean equality check in a function call, such like `f(a=b)`, use the **isequal** function or the == operator, like `f(isequal(a, b))` or `f(a == b)`.

6.6 Passing Options in any Order

We can use variable arguments along with pairs in order to pass one or more optional arguments in any order.

```
> f := proc(?) is
>    local bailout, iterations := 2, 128; # default values
>    for i to nargs do
>       case left(?[i])
>          of 'bailout' then
>             bailout := right(?[i]);
>          of 'iterations' then
>             iterations := right(?[i]);
>          else
>             print 'unknown option'
>          esle
>       esac
>    od;
>    print('bailout = ' & bailout, 'iterations = ' & iterations)
> end;

> f();
bailout = 2      iterations = 128

> f('bailout':10);
bailout = 10     iterations = 128

> f('iterations':32, 'bailout':10);
bailout = 10     iterations = 32
```

Again, the quotes around the option name (the left-hand side of the pair) can be spared by giving the = token which converts the name to a string.

```
> f(bailout = 10, iterations = 32);
bailout = 10     iterations = 32
```

Sometimes, implementing checks on options may take a substantial amount of programming time, so please have a look at the **checkoptions**, **copyadd** and the **opt*** functions which may save up to 20 % of code. You might consult Chapter 8 for further details.

6.7 Type Checking

Although Agena is untyped, in many situations you may want to check the type of a certain value passed to a function. Agena has four facilities for this:

1. the **type** operator determines the basic type of its argument;
2. the **typeof** operator returns a basic or user-defined type;
3. the **::** operator checks for a basic or user-defined type;
4. the **:-** operator checks whether a value is not of a given basic or user-defined type;

Basic or user-defined types can optionally be specified in the parameter list of a procedure by means of the preceding :: token so that they will be checked at

procedure invocation, see Chapter 6.8.2. Furthermore, the type or types of return of a procedure may be given right after the parameter list, see Chapter 6.8.3.

The following basic types are available in Agena:

```
boolean, complex, lightuserdata, null, number, pair, procedure,
register, sequence, set, string, table, thread, userdata.
```

These names are reserved keywords, but with the exception of the **null** constant evaluate to strings so that they can be compared with the result of the **type** operator:

<div style="text-align:center">

type(*value***)**

</div>

```
> type(1):
number

> type(1) = number:
true
```

If you want to check for the **null** type, put the **null** token in quotes:

```
> a := null;

> type(a) = 'null':
true
```

The **::** and **:-** operators check whether their arguments are or are not of a specific type - or user-defined type - and return **true** or **false**. They are speed-optimised and around 20 % faster than comparing the return of the **type** operator with a type name.

<div style="text-align:center">

value **::** *typename*
value **:-** *typename*

</div>

Examples:

```
> 1 :: number:
true

> '1' :- number:
true
```

In case of user-defined types, the type name must always be a string, in quotes. See Chapter 6.12 for more information. The **::** and **:-** operators can also isolate numbers further by passing the tokens `integer`, `posint`, `nonnegint`, `positive`, `negative`, or `nonnegative`, see Chapter 6.8.2 for further information.

```
> -1 :: nonnegative:
false
```

6.8 Error Handling

6.8.1 The error Function

The **error** function immediately terminates procedure execution, and prints an error message if given.

> **error(**'*error string*'**)**

```
> fact := proc(n) is
>    if n :- number then
>       error('Error: number expected')
>    fi;
>    if n < 0 then return null
>    elif n = 0 then return 1
>    else return fact(n – 1)*n
>    fi
> end;

> fact('10'):
Error: number expected

Stack traceback:
   stdin, at line 3, at line 1
```

6.8.2 Type Checks in Procedure Parameter Lists

You may specify permitted types in the parameter list of a procedure by using double colons:

```
> fact := proc(n :: number) is
>    if n < 0 then return null
>    elif n = 0 then return 1
>    else return fact(n – 1)*n
>    fi
> end;
> fact('10'):
Error in stdin:
   invalid type for argument #1: expected number, got string.
```

This form of type checking is more than twice as fast as the **if**/**type**/**error** combination. If the argument is of the correct type, Agena executes the procedure, otherwise it will issue an error. Agena will also throw an error if the argument is not given:

```
> fact()
Error in stdin:
   missing argument #1 (type number expected).
```

Finally, **argerror** is a little bit smarter than **error** for it automatically indicates the type of an argument actually passed to a procedure in its error message.

```
> a := 1;
```

```
> if a :- string then
>    argerror(a, 'myproc', 'expected a string')
> fi
Error in `myproc`: expected a string, got number.
```

Furthermore, you may specify a set of one to five permissible *basic* types for any parameter with the set notation:

```
> sec := proc(x :: {number, complex}) is
>    return 1/cos(x)
> end;
```

Besides the basic types number, complex, string, table, set, pair, sequence and register, you can also pass the following keywords to further isolate numbers:

Keyword	Check for
`integer`	a number that represents a signed integer
`posint`	a number that represents a positive integer
`nonnegint`	a number that represents a non-negative integer
`positive`	checks for a positive number (float or integer)
`negative`	checks for a negative number (float or integer)
`nonnegative`	checks for a non-negative number (float or integer)

Note that in Agena there is only one type that represents floats and integers: type number. The above mentioned five numeric `types` are only supported in parameter lists and by the **::** and **:-** operators.

Finally, there are three pseudo-types:

- `anything` stands for any type, including `'null'`. If given in a parameter list, then Agena checks whether the corresponding argument of any type, even `'null'`, has been passed in a function call - if not, an error will be issued. The pseudo-type can also be passed as the right operand to the **::** and **:-** operators;

- `listing` identifies a table, sequence or register in the parameter list of a procedure. The type can be passed as the right operand to **::** and **:-**, as well.

- `basic` identifies a number, string, Boolean or **null**, and is recognised in parameter lists and the **::** and **:-** operators.

Examples that summarise these special types:

```
> proc f(x :: anything) :: listing is
>    return x
> end;

> f()
Error in stdin:
   missing argument #1 (of type anything).

Stack traceback:
   stdin, in `f`
   stdin, at line 1 in main chunk
```

```
> f(1)
Error in stdin at line 2:
   Error in `return`: result of type listing expected, got number.

Stack traceback:
   stdin, at line 2 in `f`
   stdin, at line 1 in main chunk

> f([1]):
[1]
```

6.8.3 Checking the Type of Return of Procedures

Agena can check whether all returns of a procedure are of one given type by specifying this return type right after its parameter list.

```
> fact := proc(n :: number) :: number is
>    if n < 0 then return undefined
>    elif n = 0 then return 1
>    else return fact(n-1)*n
>    fi
> end;

> fact(10):
3628800
```

If one of the returns is not of the return type, the procedure issues an error.

```
> fact := proc(n :: number) :: number is
>    if n < 0 then return undefined
>    elif n = 0 then return 1
>    else return 'don\'t know'
>    fi
> end;

> fact(10):
Error in stdin, at line 5:
   `return` value must be of type number, got string.

Stack traceback:
   stdin, at line 5, at line 1
```

The `virtual` types `integer`, `posint`, `nonnegint`, `positive`, `negative` and `nonnegative` can also be queried, see previous subchapter.

You can define up to five basic types that are allowed to be returned by putting them in curly brackets, just like in parameter lists:

```
> f := proc(x) :: {number, complex} is return 'a' end

> f()
In stdin at line 1:
   Error in `return`: unexpected type string in return.
```

If you would like to automatically check structures for proper content at function invocation, please have a look at the end of Chapter 6.19.

There are further functions for error handling:

6.8.4 The assume Function

assume checks a Boolean relation. If the relation is valid, it returns **true** and continues execution of the procedure. In case of an invalid relation, it bails out of the procedure and prints an error message. The second argument to **assume** is optional; if not given, the text `assumption failed` is printed, and 'error string' otherwise.

<div style="text-align:center">

assume(relation [, 'error string']**)**

</div>

```
> assume(1 = 1, '1 is not 1'):
true    1 is not 1

> assume(1 <> 1, '1 is 1'):
Error in `assume`: 1 is 1.

Stack traceback: in `assume`
   stdin, at line 1 in main chunk
```

6.8.5 Trapping Errors with protect/lasterror

protect traps any error that might occur, but does not terminate a function call. In case of no errors, it returns all results of the call. But if there was an error, it returns the error message as a string and also sets the global variable **lasterror** to this error message. In case of a successful call, **lasterror** will always be **null**.

protect takes the name of the function to be executed as its first argument, and all its arguments a, b, etc. as optional arguments:

<div style="text-align:center">

protect(f [, a [, b, · · ·]]**)**

</div>

Thus, if a function has no arguments, simply pass the expression `protect(f)`.

```
> iszero := proc(x) is
>    if x <> 0 then
>       error('argument must be zero')
>    else
>       return true
>    fi
> end;
```

Now call iszero in protected mode:

```
> protect(iszero, 0):
true
```

```
> lasterror:
null

> protect(iszero, 1):
argument must be zero

> lasterror:
argument must be zero
```

To conveniently check whether an error occurred you might enter:

```
> protect(iszero, 0) = lasterror:
false

> protect(iszero, 1) = lasterror:
true
```

6.8.6 Trapping Errors with the try/catch Statement

Instead of intercepting errors with **protect** and **lasterror**, you may use the **try/catch** statement:

> **try**
>   *statements₁*
> **[catch** [*errvar* **then**]
>   *statements₂*]
> **yrt**

Any statements *statements₁* - one or more - are put right after the **try** keyword. If an error occurs in one of these statements, Agena immediately will jump to the **catch** clause if present, ignoring any subsequent statements in *statements₁*. If there is no **catch** clause, execution will immediately continue with the statement right after the **yrt** token, regardless of whether an error occurred or not, also ignoring all subsequent commands in *statements₁*.

If a **catch** clause is given, then in case of an error the error message is stored to the local variable *errvar*, and after that all the statements *statements₂* following the **then** keyword are processed. *errvar* does not need to be declared, it is implicitly local to the **catch** clause only. You may also do without specification of an error variable - in this case the error message is automatically stored to the local **lasterror** variable, and the **then** keyword must be left out.

Examples:

```
> try
>    error('Oops !');
>    print('Invalid index !')
> yrt;
```

As shown above, due to the immediate jump out of the **try** body, the **print** function is not called. In the next example, the error message is stored to the variable `message`, and in the **catch** clause it is then printed at the console.

```
> try
>    error('Oops !');
>    print('Invalid index !')
> catch message then
>    print('The error was: ' & message);
> yrt;
The error was: Oops !

> message:
null
```

Now we do not specify an error variable in the **catch** clause:

```
> try
>    error('Oops !');
>    print('Invalid index !')
> catch
>    print('The error was: ' & lasterror);
> yrt;
The error was: Oops !
```

6.8.7 Trapping Errors with pre and post clauses

Instead of writing long special error treatment code when checking arguments or the return of a function, you may use the **pre** and **post** clauses:

The **pre** clause, placed right before the **is** keyword, checks a condition and issues an error if it is not met:

```
> golden := proc(n :: number)     # approximation of golden ratio
>    pre isint(n) and n > -1 is  # if n < 0 or float, quit with an error
>    if n = 0 then return 1 fi;
>    return 1.0 + 1.0/procname(n - 1);
> end;

> golden(-1):
In stdin at line 2:
   Error in pre-condition: posture not satisfied.
```

It is faster than checking arguments with calls to the **assume** function.

The **post** clause in **return** statements checks a condition and issues an error if it is not met:

```
> proc(x :: number) is
>    [...]
>    # issue an error if x <> 1, and return x otherwise
>    return post x <> 1 with x
> end;
```

A function can include both **pre** and **post** conditions.

6.9 Multiple Returns

As stated before, a procedure can return no, one, or more values. Just specify the values to be returned:

```
> f := proc() is
>    a := 2;
>    return 1, a
> end;

> f():
1        2
```

There are two ways to refer to these multiple returns in subsequent statements. If you assign the return to only one variable, e.g.

```
> m := f():
1
```

the second return is lost, so enter:

```
> m, n := f();

> m:
1

> n:
2
```

A function may return a variable number of values, so it might be useful to put them in a sequence, register or table:

```
> seq(f()):
seq(1, 2)
```

Sometimes a procedure shall return the first result of a computation only. In this case, put the call that results into multiple returns into brackets. **math.fraction** returns three values: the numerator, the denominator, and the accuracy, in this order. Let us write a numerator function that only returns the first result of **math.fraction**.

```
> numerator := proc(x :: number) is
>    return (math.fraction(x))
> end;
> numerator(0.1):
1
```

The **ops** function returns all its arguments after argument number index, an integer.

$$\boxed{\textbf{ops(}\text{index, arg}_1 \text{ [, arg}_2, \cdots \text{])}}$$

The following statement determines the denominator and the accuracy.

```
> ops(2, math.fraction(0.1)):
10       0
```

To return only the first result, the denominator, put the call to **ops** in brackets.

```
> denominator := proc(x :: number) is
>    return (ops(2, math.fraction(x)))
> end;

> denominator(0.1):
10
```

unpack returns all elements in a table or sequence:

```
> squared := proc(t :: table) is
>    local result := << x -> x^2 >> @ t;
>    return unpack(result)
> end;

> squared([1, 2, 3, 4]):
1       4       9       16
```

Alternatively, **unpack** accepts the positions of the first to the last element to be returned as its second and third argument. If only the second argument is given, all elements in a structure from the given position up to the end are passed back.

<div style="border:1px solid black; text-align:center;">

unpack(structure [, beginning [, end]] **)**

</div>

```
> squared := proc(t :: table, ?) is
>    local result := << x -> x^2 >> @ t;
>    return unpack(result, unpack(?))
> end;

> squared([1, 2, 3, 4], 2):
4       9       16

> squared([1, 2, 3, 4], 2, 3):
4       9
```

6.10 Procedures that Return Procedures

Besides returning numbers, strings, tables, etc., procedures can also return procedures. As an example, the function `polygen`

```
> polygen := proc(?) is
>    local s := seq(unpack(?));
>    return proc(x) is
>        local r := bottom(s);
>        for i from 2 to size s do
>            r := r*x + s[i]
>        od;
>        return r
>    end
> end;
```

returns a procedure that evaluates a polynomial of degree n from the given coefficients $c_n, c_{n-1}, \cdots, c_2, c_1$:

$$<< \; (x) \; -> \; c_n{}^{*}x^{n-1} + c_{n-1}{}^{*}x^{n-2} + \cdots + c_2{}^{*}x + c_1 \; >>$$

In the following example, `polygen` creates the polynomial $3x^2 - 4x + 1$ as a procedure.

```
> f := polygen(3, -4, 1)

> f(2):
5
```

6.11 Shortcut Procedure Definition

If your procedure consists of exactly one *expression*, then you may use an abridged syntax if the procedure does not include statements such as **if/then**, **for**, **insert**, etc.

> $<<$ [(] [*par₁* [:: *type₁*] [, *par₂* [:: *type₂*], $\cdots$]] [)] -> expr₁ [, expr₂, $\cdots$] $>>$
>
> $<<$ [(] [*par₁* [:: *type₁*] [, *par₂* [:: *type₂*], $\cdots$]] [)]
> [**with** var1 [, $\cdots$] := val1 [, $\cdots$]] -> expr₁ [, expr₂, $\cdots$] $>>$

As you see, optional basic and user-defined types can be specified in the parameter section.

Let us define a simple factorial function.

```
> fact := << (x :: number) -> exp(lngamma(x + 1)) >>;

> fact(4):
24
```

Brackets around parameters are optional if at least one parameter is given, even if you specify types.

```
> isInteger := << x -> int(x) = x >>;

> isInteger(1):
true

> isInteger(1.5):
false

> one := << () -> 1 >>;  # with no parameters, use empty bracket pair
```

Optional arguments and the ? notation are supported.

One or more local variables can be defined by the **with** clause put in front of the expression that computes the result:

```
> fact := << (x :: number)
>    with n := 1
>    -> exp(lngamma(x + n)) >>;

> fact(4):
24
```

Short-cut procedures can return multiple results:

```
> f := << x -> x, x+1, x+2 >>

> f(0):
0        1        2
```

6.12 User-Defined Procedure Types

The **settype** function allows to group procedures $proc_1$, $proc_2$, $\cdots$, by giving them a specific type (passed as a string) just as it does with sequences, tables, sets, and pairs.

> **settype(**$proc_1$ [, $proc_2$, $\cdots$], 'your_proctype'**)**

User-defined procedures can be queried with the **typeof** operator which returns a string.

```
> f := << x -> 1 >>;

> settype(f, 'constant');

> typeof(f):
constant

> type(f):  # only returns the basic type
procedure
```

The **::** and **:-** operators can also validate a user-defined procedure type. Pass the name of the user- defined type as a string:

> $proc_1$ **::** 'your_proctype'
> $proc_1$:- 'your_proctype'

```
> f :: 'constant':
true

> f :- 'constant':
false
```

Note that the **type** operator only checks for basic types.

An alternative to **typeof** is the **gettype** function. If a user-defined type has set for a value, then it returns its name as a string, otherwise, it returns **null**.

If you want to check whether user-defined types have been passed to a procedure, use the double colon notation in the parameter list.

Suppose you have defined a type called `triple`:

```
> t := [1, 2, 3]

> settype(t, 'triple')

> sum := proc(x :: triple) is
>    return sumup(x)
> end

> sum(t):
6
```

6.13 Scoping Rules

In Agena, variables live in blocks or `scopes`. A block may contain one or more other blocks. A local variable is visible only to the block in which it has been declared and to all blocks that are part of this block. Thus, variables declared local in inner blocks are not accessible to the outer blocks or outside the procedure in which they are hosted.

Procedures, **if**- and **case**-statements, **while**-, **do**- and **for**-loops create blocks, or more precisely, a block resides between:

1. **then** and **elif**, **else** or **fi** keywords - in **if** statements;
2. **then** and **of**, **else** or **esac** keywords - in **case** statements;
3. **do** and **as** - in **do/as** loops;
4. **do** and **od** - in **for** and **while** and **do/od** loops;
5. **is** and **end** - in procedures;
6. **scope** and **epocs** - in **scope** blocks (including the **with** statement; see below).

As an example, variables declared as local in the **then** clauses of an **if**-statement live only in the respective **then** part. The same applies to variables declared locally in **else** clauses.

```
> f := proc(x) is
>    if x > 0 then
>       local i := 1; print('inner', i)
>    else
>       local i := 0; print('inner', i)
>    fi;
>    print('outer', i)  # i is not visible
> end;
> f(1);
inner   1
outer   null
```

Variables declared as local in **for**- or **while**-loops are only accessible in the bodies of these loops. The loop control variables of **for/to**-loops are automatically declared local to their surrounding block, while control variables of **for/in**-loops are implicitly declared local to the respective loop bodies.

```
> f := proc(x) is
>    while x < 2 do
>       local i := x
>       inc x
>       print('inner', i)
>    od;
>    print('outer', i)  # i is not visible
> end;

> f(1);
inner   1
outer   null
```

A special scope can be declared with the **scope** and **epocs** statements:

scope
   declarations & statements
epocs

The next example demonstrates how it works:

```
> f := proc() is
>    local a := 1;
>    scope
>       local a := 2;
>       writeline('inner a: ', a);
>    epocs;
>    writeline('outer a: ', a);
> end;

> f()
inner a: 2
outer a: 1
```

The **scope** statement can also be used on the interactive level to execute a sequence of statements as one unit. Compare

```
> print(1);
1

> print(2);
2

> print(3);
3
```

with

```
> scope
>    print(1);
>    print(2);
>    print(3)
> epocs;
1
2
3
```

6.14 Access to Loop Control Variables within Procedures

As already mentioned, the control variable of a **for/to** loop is always local to the body surrounding the loop.

```
> mandelbrot := proc(x, y, iter, radius) is
>    local i, c, z;
>    z := x!y;
>    c := z;
>    for i from 0 to iter while abs(z) < radius do
>        z := z squareadd c  # = z^2 + c
>    od;
>    return i  # return the last iteration value
> end;
```

The procedure counts and returns the number of iterations a complex value z takes to escape a given radius by applying it to the formula $z = z^2+c$.

```
> mandelbrot(0, 0, 128, 2):
129
```

The following example demonstrates that local variables are bound to the block in which they have been declared.

```
> f := proc() is
>    local i;
>    for i to 3 do
>        local j;
>        for j to 3 do od;
>        print(i, j)
>    od;
>    print(i, j)
> end;

> f()
1       4
2       4
3       4
4       null
```

6.15 Sandboxes

By default, every procedure has access to the full Agena environment, i.e. to all of Agena's functions, packages, and all the other values. You might want to limit this access, for example if one of your procedures offers services on the Internet, or you want a procedure maintain its own environment.

Here, the **environ.setfenv** function comes into play. It initialises the environment a function can use.

Example 1: Give access to all functions except the **os** package.

First copy Agena's environment represented by the system table **_G** to a new table so that altering this new table will not effect Agena's normal environment:

```
> _newG := copy(_G);  # copy can also duplicate cycles like _G
```

Delete the **os** package from this new environment:

```
> delete os from _newG;
```

Define a function that tries to determine the current working directory:

```
> curdir := proc() is
>    return os.chdir()
> end;
```

Set the environment excluding the **os** package:

```
> environ.setfenv(curdir, _newG);

> curdir():
Error in stdin, at line 2:
   attempt to index global `os` (a null value) with a string value

Stack traceback:
   stdin, at line 2, at line 1
```

Example 2: Give access only the specific functions.

Let us redefine curdir: it will only access a redefined **print** function and all of the functions of the **os** package. curdir cannot call any other function.

```
> curdir := proc() is
>    print(os.chdir())
> end;

> environ.setfenv(curdir,
>    ['print' ~ << x -> print('cwd is ' & x) >>, 'os' ~ os])

> curdir():
cwd is C:/agena/src
```

To determine the current environment used by a function, use **environ.getfenv**:

```
> environ.getfenv(curdir):
[os ~ (···), print ~ procedure(01D4BA18)]
```

Please see Chapter 14.2 (**environ.getfenv**, **environ.setfenv**, **environ.isselfref**) for further features.

To hide data in a sandbox, please have a look at registers - explained in Chapter 4.15.

6.16 Altering the Environment at Run-Time

Besides using a special environment (see preceding subchapter), a procedure can also create new variables and put them into Agena's standard environment.

Why should one do so ? Consider the **utils.decodexml** function. It converts an XML
string into a table consisting of key-value pairs, the keys being the XML tags, and the
values the corresponding data. XML allows to use name spaces, so that tags might
look like `<soap:body>`, etc.

So, XML data like

```
> str := '<soap:body>
>    <orderid>123</orderid>
> </soap:body>'
```

is converted to

```
> order := utils.decodexml(str):
[soap_body ~ [orderid ~ 123]]
```

To read the order number, one might just enter:

```
> order.soap_body.orderid:
123
```

Unfortunately, especially the SOAP standard allows one to define ones own name
space, so that the following is also equivalent and valid XML data:

```
> str := '<s:body>
>    <orderid>123</orderid>
> </s:body>'

> order := utils.decodexml(str):
[s_body ~ [orderid ~ 123]]
```

In this case you would have to write a new statement to get the order ID since
fetching it with

```
> order.soap_body.orderid:
Error in stdin, at line 1:
   attempt to index field `soap_body` (a null value)
```

will not work. Fortunately, Agena stores all values in the **_G** system table, with its keys
being strings representing the variable names, and the entries the values of the
these variables. So flexible code to read data from XML code featuring different
name spaces might look like this:

```
> str := '<s:body>
>    <orderid>123</orderid>
> </s:body>'

> order := utils.decodexml(str):
[s_body ~ [orderid ~ 123]]

> tag := tables.indices(order)[1]:
s_body

> prefix := tag[1 to ('_' in tag) − 1]:
s
```

```
> _G['order'][prefix & '_body'].orderid:
123
```

Likewise, defining new variables within code can be done like this:

```
> _G['jpl'] := ['Jet Propulsion Laboratory']

> jpl:
[Jet Propulsion Laboratory]
```

6.17 Packages

6.17.1 Writing a New Package

Let us write a small utilities package called `helpers` including only one main and one auxiliary function. The main function shall return the number of digits of an integer.

Package procedures are usually stored to a table, so we first create a table called `helpers`. After that, we assign the procedure `ndigits` and the auxiliary `aux.isInteger` function to this table.

```
> create table helpers, helpers.aux;

> helpers.aux.isInteger := << x -> int(x) = x >>;  # aux function

> helpers.ndigits := proc(n :: number) is
>    if not helpers.aux.isInteger(n) then
>       error('Error, argument is not an integer')
>    fi;
>    return if n = 0 then 1 else entier(ln(abs(n))/ln(10) + 1) fi
> end;
```

Now we can use our new package.

```
> helpers.ndigits(0):
1

> helpers.ndigits(-10):
2

> helpers.ndigits(.1):
Error, argument is not an integer

Stack traceback: in `error`
   stdin, at line 3, at line 1
```

To save us a lot of typing, we can assign a short name to this table procedure.

```
> ndigits := helpers.ndigits;

> ndigits(999):
3
```

Save the code listed above to a file called `helpers.agn` in a subfolder called `helpers` in the Agena main directory. In order to use the package again after you

have restarted Agena, use the **run** function and specify the full path.

```
> restart;

> run 'd:/agena/helpers/helpers.agn'

> helpers.ndigits(10):
2
```

You may print the contents of the package table at any time:

```
> helpers:
[aux ~ [isInteger ~ procedure(0044A6E0)], ndigits ~ procedure(0044A850)]
```

6.17.2 The initialise Function

The **initialise** function, besides loading the package in a convenient way, automatically assigns short names to all package procedures so that you may use the shortcuts instead of the fully written function names.

In order to do this, you must first prepend or append the location of the directory containing your new package to the **libname** system variable, or execute Agena in the directory containing your package. You may do this by adding the following line to your personal Agena initialisation file (see Chapter A6), assuming that the `helpers.agn` file has been stored to the folder `d:/agena/helpers`.

```
libname &:= ';d:/agena/helpers';
```

Alternatively, you may save the helpers.agn file into the `lib` folder of your Agena distribution if you do not want to modify **libname**.

Now in the interactive level, type:

```
> restart;
```

libname and some few other system variables are not reset by the **restart** statement because **restart** deliberately does not touch the contents of these specific system variables.

```
> initialise 'helpers'
ndigits

> ndigits(1); # same as helpers.ndigits(1)
```

You may also want **with** to print a start-up notice at every package invocation by assigning a string to the table field `packagename.**initstring**`. Put the following line into the `helpers.agn` file after the **create table** statement, save the file and restart Agena:

```
> helpers.initstring := 'helpers v1.0 as of June 11, 2013\n\n';
```

```
> restart;

> initialise 'helpers'
helpers v1.0 as of June 11, 2013

ndigits
```

Since you may not want that short names are set for certain, especially auxiliary functions, their procedure names should be defined as follows: `packagename.**aux**.procedurename`, e.g. `helpers.aux.isInteger`.

The contents of the `helpers.agn` file should finally look like this:

```
create table helpers, table helpers.aux;

helpers.initstring := 'helpers v1.0 as of June 11, 2013\n\n';

helpers.aux.isInteger := << x -> int(x) = x >>;  # aux function

helpers.ndigits := proc(n :: number) is
   if not helpers.aux.isInteger(n) then
      error('argument is not an integer')
   fi;
   if n = 0 then
      return 1
   else
      return entier(ln(abs(n))/ln(10) + 1);
   fi;
end;
```

Save the file again and restart Agena.

```
> restart;

> initialise 'helpers'
helpers v1.0 as of June 11, 2013

ndigits
```

You can also define a package initialisation routine. It will automatically be run by the **initialise** statement after the package has been found and initialised successfully. The name of the initialisation routine must be of the form `packagename.**aux.init**`, e.g.:

```
> helpers.aux.init := proc() is
>    writeline('I am being run')
> end;
```

Of course, you must create a `packagename.**aux**` table before defining the initialisation function.

Instead of using **initialise** to load a package, you may use the **import/alias** statement - see Chapter 3.18 - so

```
> initialise 'helpers';
```

is equivalent to

```
> import helpers alias;
```

6.18 Remember Tables

Agena features remember tables which store the results of previous calls to Agena or C library procedures or contain a list of predefined results, or both. If a function is called again with the same argument(s), then the corresponding result will be returned from the table, and the procedure body is not executed, resulting in significantly better execution times. Remember tables are called *rtables* or *rotables* for short.

All functions to create, modify, query, and delete remember tables are available in the **rtable** package.

There are two types of remember tables:

• Standard Remember Tables, called `rtables`, that can be automatically updated by a call to the respective function; they may be initialised with a list of precomputed results (but do not need to).

• Read-only Remember Tables, called `rotables`, that cannot be updated by a call to the respective function. Rotables should be initialised with a list of precomputed results.

6.18.1 Standard Remember Tables

A standard remember table is suited especially for recursively defined functions. It may slow down functions, however, if they have remember tables but do not rely much on previously computed results.

By default, no procedure contains a remember table. It must explicitly be created either by including the **feature reminisce** statement as the very first line in a procedure body, or by calling the **rtable.init** function right after the procedure has been defined. A remember table may optionally be filled with default values with the **rtable.put** function. Since those functions are very basic, a more convenient facility is the **rtable.remember** function which will exclusively be used in this chapter.

In order for an rtable to be automatically updated, the respective function must return its result with the **return** statement (which may sound profane). If a function is called with arguments that are not already known to the remember table, then the **return** statement adds these arguments and the corresponding result or results to the rtable.

Let us first try the **feature reminisce** variant, which may suffice in most cases. Just add this statement right after the **is** token in a procedure that computes Fibonacci numbers:

```
> fib := proc(n) is
>    feature reminisce;  # creates a read-write remember table
>    if n = 0 or n = 1 then return n fi;  # exit conditions
>    return procname(n - 2) + procname(n - 1)
> end;

> fib(50):
20365011074
```

Now let us use the functions of the **rtable** package to administer remember tables.

Two examples: We want to define a function f(x) = x with f(0) = undefined.

First a new function is defined without using the **feature reminisce** phrase:

```
> f := proc(x) is return x end;
```

Only after the function has been created in such a way, the remember table can be set up. The **rtable.remember** function can be used to initialise rtables, explicitly set predefined values into them, and add further values later in a session.

```
> import rtable alias;

> remember(f, [0 ~ undefined]);
```

The rtable has now been created and a default entry add to it so that calling f with argument 0 returns **undefined** and not 0.

```
> f(1):
1

> f(0):
undefined
```

If the function is redefined, its remember table is destroyed, so you may have to initialise it again.

Fibonacci numbers, as already shown above, can be implemented recursively and run with astonishing speed using rtables.

```
> fib := proc(n) is
>    assume(n >= 0);
>    return procname(n - 2) + procname(n - 1)
> end;
```

The call to **assume** assures that n is always non-negative and serves as an `emergency brake` in case the remember table has not been set up properly.

The rtable is being created with two default values:

```
> remember(fib, [0~0, 1~1]);
```

If we now call the function,

```
> fib(50):
20365011074
```

the contents of the rtable will be:

```
> remember(fib):
[[22] ~ [28657], [39] ~ [102334155], [17] ~ [2584], [5] ~ [8], [27] ~
[317811], [50] ~ [20365011074], [3] ~ [3], [0] ~ [1], [46] ~ [2971215073],
[41] ~ [267914296], [1] ~ [1], etc.]
```

If a function has more than one parameter or has more than one return, **remember** requires a different syntax: The arguments and the returns are still passed as key~value pairs. However, the arguments are passed in one table, and the returns are passed in another table.

```
> f := proc(x, y) is
>    return x, y
> end;

> remember(f, [[1, 2] ~ [0, 0]]);

> a, b := f(1, 2);

> a:
0

> b:
0
```

Please check Chapter 14.4 for more details on their use.

6.18.2 Read-Only Remember Tables

If you do not want a function updating its remember table each time it is called with new arguments and results, you may use a read-only remember table, called `rotable` for short. Rotables are initialised with a list of precomputed results.

The function itself cannot implicitly enter new entries to its remember table via the **return** statement; it can only do so via a call to the **rtable.put** function or a utility that is based on **rtable.put**, called **rtable.defaults**. This gives you full control on the contents and the amount of data stored in a remember table - and thus on the speed of your procedure.

Assume you want to define a procedure that computes factorials n!, and that does not compute the results for n < 11, but retrieves the results from an rotable instead.

A function might look like this:

```
> fact := proc(x :: number) is
>    if x :: nonnegint then  # is x an integer and non-negative ?
>       return exp(lngamma(x + 1))
>    else
>       return undefined
>    fi
> end;
```

The **defaults** function can set up the rotable and enter precomputed values into it.

```
> # set precompiled results for 0! to 10! to fact

> defaults(fact, [
>    0~1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800
>    ]);
```

The factorial function is significantly faster when called with arguments that are in the rotable than if there would be no such value cache, because it would have to re-compute the results instead of just reading them.

Let us look into the remember table:

```
> defaults(fact):
[[2] ~ [2], [1] ~ [1], [8] ~ [40320], [9] ~ [362880], [10] ~ [3628800],
[0] ~ [1], [4] ~ [24], [5] ~ [120], [6] ~ [720], [3] ~ [6], [7] ~ [5040]]
```

You can also easily add further argument ~ result pairs with the **rtable.defaults** function:

```
> defaults(fact, [11 ~ 39916800]);

> defaults(fact):
[[2] ~ [2], [1] ~ [1], [8] ~ [40320], [9] ~ [362880], [10] ~ [3628800], [0]
~ [1], [11] ~ [39916800], [4] ~ [24], [7] ~ [5040], [6] ~ [720], [3] ~ [6],
[5] ~ [120]]
```

A read-only remember table can be deleted by passing **null** as a second argument to **defaults**.

Please note that in a function featuring a remember table, the respective **return** statements should not include calls to other functions than the function itself. Instead, use auxiliary variables and use them in the **return** statements.

6.18.3 Functions for Remember Table Administration

For completeness, here is a list of all the functions to administer remember tables:

Procedure	Details
rtable.forget(f)	Empties the remember table of function `f` but does not delete the table so that it will continue collecting results with the next call to f. Read-only remember tables cannot be emptied. The memory previously occupied by cached function arguments and results can be reused for other purposes
rtable.get(f)	Returns the remember table of function `f`.
rtable.init(f)	Initialises a standard remember table for the function `f`.
rtable.roinit(f)	Initialises a read-only remember table for the function `f`.
rtable.put(f, **[**arguments**]**, **[**returns**])**	Adds function argument(s) and the corresponding return(s) to the remember table of procedure `f`.
rtable.purge(f)	Deletes the remember table of function `f` entirely. The function empties the remember table before deleting it. It also enforces an immediate garbage collection. If you want to use a new remember table with the function, you have to initialise it with **rtable.init** or **rtable.roint** again.
rtable.mode(f)	Returns the string 'rtable' if a function `f` has a standard remember table, 'rotable' if it has a read-only remember table, and 'none' if it has no remember table at all.

Table 18: Functions for administering remember tables

6.19 Overloading Operators with Metamethods

One of the many useful functions inherited from Lua 5.1 are metamethods which provide a means to use existing operators to tables, sets, sequences, registers, pairs, and userdata.

For example, complex arithmetic could be entirely implemented with metamethods so that you can use already existing symbols and keywords such as + or **abs** with complex values and do not have to learn names of new functions[16]. This method of defining additional functionality to existing operators is also known as `overloading`.

Adding such functionality to existing operators is very easy. As an example, we will define a constructor to produce complex values and three metamethods for adding complex values with the + token, determining their absolute value with the standard **abs** operator, and pretty printing them at the console.

---

[16] For performance reasons, complex arithmetic has been built directly into the Agena kernel.

At first, let's store a complex value z = x + y*i* to a sequence of size 2. The real part is saved as the first value, and the imaginary part as the second.

```
> cmplx := proc(a :: number, b :: number) is
>    create local sequence r(2);
>    insert a, b into r;
>    return r
> end;
```

To define a complex value, say z = 0 + *i*, just call the constructor:

```
> cmplx(0, 1):
seq(0, 1)
```

The output is not that nice, so we would like Agena to print `cmplx(0, 1)` instead of `seq(0, 1)`. This can be easily done with the **settype** function:

```
> cmplx := proc(a :: number, b :: number) is
>    create local sequence r(2);
>    insert a, b into r;
>    settype(r, 'cmplx');
>    return r
> end;

> cmplx(0, 1):
cmplx(0, 1)
```

Adding two complex values does not work yet, for we have not yet defined a proper metamethod.

```
> cmplx(0, 1) + cmplx(1, 0):
Error in stdin, at line 1:
   attempt to perform arithmetic on a sequence value
```

Metamethods are defined using dictionaries, called `metatables`. Their keys, which are always strings, denote the operators to be overloaded, the corresponding values are the procedures to be called when the operators are applied to tables, sets, sequences (which are used in this example), registers or pairs. See Appendix A2 for a list of all available method names. To overload the plus operator use the '`__add`' string.

Assign this metamethod to any name, `cmplx_mt` in this example.

```
> cmplx_mt := [
>    '__add' ~ proc(a, b) is
>                return cmplx(a[1]+b[1], a[2]+b[2])
>             end
> ]
```

Next, we must attach this metatable `cmplx_mt` to the sequence storing the real and imaginary parts with the **setmetatable** function. We have to extend the constructor by one line, the call to **setmetatable**:

```
> cmplx := proc(a :: number, b :: number) is
>    create local sequence r(2);
>    insert a, b into r;
>    settype(r, 'cmplx');
>    setmetatable(r, cmplx_mt);
>    return r
> end;
```

Try it:

```
> cmplx(0, 1) + cmplx(0, 1):
cmplx(0, 2)
```

Add a new method to calculate the absolute value of complex numbers by overloading the **abs** operator.

```
> cmplx_mt.__abs := << (a) -> hypot(a[1], a[2]) >>;
```

The metatable now contains two methods.

```
> cmplx_mt:
[__add ~ procedure(004A64D0), __abs ~ procedure(004D2D30)]

> z := cmplx(1, 1);

> abs(z):
1.4142135623731
```

It would be quite fine if complex values would be output the usual way using the standard x + y*i* notation. This can be done with the '$\_\_tostring$' method which must return a string.

```
> cmplx_mt.__tostring := proc(z) is
>    return if z[2]<0 then z[1]&z[2]&'i' else z[1]&'+'&z[2]&'i' fi
> end;
> z:
1+1i
```

To avoid using the **cmplx** constructor in calculations, we want to define the imaginary unit I = 0+i and use it in subsequent operations. Before assigning the imaginary unit, we have to add a metamethod for multiplying a number by a complex number.

```
> cmplx_mt.__mul := proc(a, b) is
>    if typeof(a) = 'cmplx' and typeof(b) = 'cmplx' then
>       return cmplx(a[1]*b[1]-a[2]*b[2], a[1]*b[2]+a[2]*b[1])
>    elif type(a) = number and typeof(b) = 'cmplx' then
>       return cmplx(a*b[1], a*b[2])
>    fi
> end;
```

and also extend the metamethod for complex addition.

```
> cmplx_mt.__add := proc(a, b) is
>    if typeof(a) = 'cmplx' and typeof(b) = 'cmplx' then
>        return cmplx(a[1]+b[1], a[2]+b[2])
>    elif type(a) = number and typeof(b) = 'cmplx' then
>        return cmplx(a+b[1], b[2])
>    fi;
> end;

> i := cmplx(0, 1);

> a := 1+2*i:
1+2i
```

Until now, the real and imaginary parts can only be accessed using indexed names, say `z[1]` for the real part and `z[2]` for the imaginary part. A more convenient - albeit not that performant - way to use a notation like `z.re` and `z.im` in both read and write operations is provided by the '`__index`' and '`__writeindex`' metamethods, respectively.

The __index metamethod for *reading* values from a structure obj works as follows:

*   If the structure is a table, then Agena will automatically call the metamethod if the lookup obj[key] results to **null**.
*   If the structure is a set, then Agena will automatically call the metamethod if the lookup obj[key] results to **false**.
*   If the structure is a sequence or register, then the metamethod is called if the lookup obj[key] would result to an index-out-of-range error.

The '`__writeindex`' metamethod for *writing* values to a structure works as follows:

*   If the structure is a table, sequence or pair, then the metamethod is always called.
*   The metamethod is also supported by the **insert** statement.

The procedures assigned to the '`__index`' and '`__writeindex`' keys of a metatable should not include calls to indexed names, for in some cases this would lead to stack overflows due to recursion (the respective metamethod is called again and again). Instead, use the **rawget** function to directly read values from a structure, and the **rawset** function to add values into a structure.

Let us first define a global mapping table for symbolic names to integer keys:

```
> cmplx_indexing := [re ~ 1, im ~ 2];
```

Now let us define the two new metamethods. Both will accept expressions like `a.re` and `a[1]`. In the following read procedure the argument `x` represents the complex value, and the argument `y` is assigned either the string `'re'` or `'im'`. Thus, `cmplx_indexing['re']` will evaluate to the index 1, and `cmplx_indexing['im']` to index 2.

```
> cmplx_mt.__index := proc(x, y) is  # read operation
>    if type(y) = string then  # for calls like `a.re` or `a.im`
>        return rawget(x, cmplx_indexing[y])
>    else
>        return rawget(x, y)    # for calls like `a[1]` or `a[2]`
>    fi
> end;
```

In the write procedure, argument `x` will hold the complex value, `y` will be either `'re'` or `'im'`, and `z` is assigned the component - a rational number -, i.e. `x.re := z` or `x.im := z`.

```
> cmplx_mt.__writeindex := proc(x, y, z) is  # write operation
>    if type(y) = string then
>        rawset(x, cmplx_indexing[y], z)
>    else
>        rawset(x, y, z)  # for assignments like `a[1] := value`
>    fi
> end;
```

You can now use the new methods.

```
> a:
1+2i

> a.re:
1

> a.im := 3;

> a:
1+3i
```

Note that while arithmetic metamethods can be applied on mixed types, for example the above defined complex number and a simple Agena number, relational operators cannot compare values of different types. Instead, Agena in this case just returns **false** with the equality operators =, ==, and ~=; and issues an error with relational operators that compare for order.

Using the '`__writeindex`' metamethod, it is quite easy to write-protect structures. In the following example, we will create a procedure that accepts a table, write-protects it and returns it. The metamethod:

```
> readonly_mt := [
>    '__writeindex' ~
>        proc(t, k, v) is error('Error, structure is read-only.') end
> ]
```

A constructor that simplifies creating read-only structures:

```
> readonly := proc(t :: table) is
>    setmetatable(t, readonly_mt);
>    return t
> end;

> moons := readonly(['Phobos', 'Deimos']);
```

Adding further values to the table, or changing an existing one, now will not work.

```
> insert 'Mars' into moons;
Error, structure is read-only.

Stack traceback: in `error`

> moons:
[Phobos, Deimos]
```

Using one and the same global table to define metamethods for various variables may be appropriate to save memory, but modification of the metatable itself may have unwanted effects.

```
> readonly_mt.__writeindex := proc(t, k, v) is rawset(t, k, v) end;

> insert 'Mars' into moons;

> moons:
[1 ~ Phobos, 2 ~ Deimos, Mars ~ Mars]
```

Finally, to protect values already assigned to a table, we could define:

```
> readonly_mt := [
>    __writeindex =
>       proc(t, k, v) is
>          if rawget(t, k) <> null then
>             error('Error, structure is read-only.')
>          else
>             rawset(t, k, v)
>          fi
>       end
> ]

> create table t;

> setmetatable(t, readonly_mt)

> t[1] := 0

> t[1] := 1
Error, structure is read-only.
```

To protect metatables from tampering, use the **__metatable** method and set it to any value except **null**.

```
> readonly_mt := [
>    __metatable = false,
>    __writeindex =
>       proc(t, k, v) is error('Error, table is read-only') end
> ];

> readonly := proc(t :: table) is
>    setmetatable(t, readonly_mt);
>    return t
> end;

> moons := readonly(['Phobos', 'Deimos']);
```

```
> setmetatable(moons, [
>    __writeindex =
>          proc(t, k, v) is error('Error, table is read-only') end
>    ]
> );
Error in `setmetatable`: cannot change a protected metatable.

Stack traceback: in `setmetatable`
   stdin, at line 1 in main chunk
```

A structure with a '$\_\_call$' key in its metatable can also be called like a function.

```
> readonly := proc(t :: table) is
>    setmetatable(t, [
>       __call = proc(t) is
>          for i, j in t do print(i, j) od
>       end]);
>    return t
> end;

> moons := readonly(['Phobos', 'Deimos']);

> moons();
1       Phobos
2       Deimos
```

To close this chapter, metamethods can also be used to automatically check the contents of structure passed at function invocation, and also to extend the :: and :-operators.

Let us assume we would like to write a procedure that sums up all numbers in a set:

```
> s := {1, 2, 3, 4, 5};
```

We create a metatable first,

```
> create table mt;
```

and then assign a proper evaluation procedure to the $\_\_oftype$ metamethod that makes sure that the set consists of numbers only.

```
> mt.__oftype := proc(x) is
>    if type x = set then
>       for i in x do
>          if i :- number then return false fi
>       od;
>       return true
>    else
>       return false
>    fi
> end
```

We assign the metatable to the set,

```
> setmetatable(s, mt);
```

and first try out the extended **::** and **:-** operators.

```
> s :: set:
true
```

If an invalid member is inserted into the set,

```
> insert 'a' into s;
```

the type check fails:

```
> s :: set:
false

> s :- set:
true
```

Now we use the type evaluator in a procedure call:

```
> sum := proc(x :: set) is
>    local s := 0; for i in x do inc s, i od; return s
> end;

> sum(s):
In stdin:
   argument #1 does not satisfy type check metamethod
```

The '__oftype' metamethod works as follows: it first checks whether the structure (a table, set, sequence, register, pair) or userdata at the left-hand side matches the basic or user-defined type given at the right-hand side. If true, then Agena checks whether the structure has an attached '__oftype' metamethod and then runs it. The validator function must either return **true** if the criteria have all been met, or **false**, **fail** or **null** otherwise.

Note that in the validator mt.__oftype definition given above, we use the **type** operator instead of the **::** operator in the first **if** statement since otherwise Agena would issue a stack overflow error.

The __oftype metamethods also work if a return type has been specified.

In some packages, for example llist and numarray, metamethods are included in the binary C library file and can be accessed through the so-called registry, via the **debug.getregistry** function. You may want to use this function to add further self-defined metamethods written in the Agena language.

For example, the __in metamethod of the numarray package is defined in the Agena source file lib/numarray.agn, and not in the C library file.

```
> numarray.aux.mt := [
>   __in = proc(x, a) is
>          return numarray.whereis(x, a, 1, Eps) <> null
>        end
> ]
```

The metatable stored to the registry can be read by a call to **registry.get**. Just insert all of your own metamethod procedures by individually adding them, but do not directly assign your metamethod table to the result of **registry.get('numarray')**.

```
> scope
>    # protect against sandboxing (prevent errors at initialisation)
>    if registry.get :: procedure then
>        # get the internal registry metatable for numarrays
>        local _mt := registry.get('numarray');
>        if _mt :: table then
>            # include each metamethod function step-by-step
>            for i, j in numarray.aux.mt do
>                _mt[i] := j
>            od
>        fi
>    fi
> epocs;
```

Never modify or delete existing metamethods, as this will lead to undefined behaviour.

Note: The **delete** statement supports metamethods: it passes the data to be deleted as its key and **null** as the value to the __writeindex metamethod. To protect values stored to structures you might define:

```
> readonly_mt.__writeindex := proc(t, k, v) is
>    if unassigned v or assigned rawget(t, k) then
>        error('cannot delete or modify value')
>    else
>        rawset(t, k, v)
>    fi
> end;
```

The **pop**, **rotate**, **duplicate**, and **exchange** statements issue an error if a given structure features a __writeindex metamethod. This prevents read-only structures from being modified.

6.20 Memory Management, Garbage Collection, and Weak Structures

Agena includes a garbage collector that sweeps all structures, procedures, userdata, and threads (called `objects` in this subchapter) that no longer have valid references in your programme - i.e. are inaccessible. Agena can then use the space for new objects. Numbers, complex numbers, strings and booleans are never collected.

Consider the following code: Let us assign a table to a name.

```
> s := []
```

Now s refers to a memory address so that Agena can access the table.

```
> environ.pointer(s):
008F0F38
```

If we reassign s, a different empty table is assigned to it.

```
> s := []
```

This newly created table is stored to another part of the memory.

```
> environ.pointer(s):
008A4188
```

Since the first table at memory position 008F0F38 can no longer be accessed, it unnecessarily occupies space. The garbage collector regularly looks for unreferenced objects and removes them.

Besides automatic garbage collection, the user can also invoke it manually, if deemed necessary, or even stop and restart it by calling **environ.gc**.

Sometimes it may be necessary to immediately clear values occupying a large amount of space. In this case assign **null** to it, so that the next automatic collection cycle can free it. If necessary call **environ.gc** for immediate collection. As a shortcut, you could also use the **clear** statement which conducts both **null**ing a value and collecting it.

If a table, set, sequence, procedure, userdata or thread is included in another table or sequence, the garbage collector does not collect it if its reference should have become invalid.

```
> restart

> t := []

> v := [1]; insert v into t

> v := [2]; insert v into t

> environ.gc()
```

[1] is still part of the table.

```
> t:
[[1], [2]]
```

If you do not want this to happen, declare the table or sequence `weak` by using the '__weak' metamethod. With tables, you can either declare its keys weak by passing the string 'k', or its values weak with the string 'v', or both with 'kv'. With sequences, simply use use the string 'v'.

If the collector meets a weak key that has become inaccessible, it removes the key-value pair. If the collector meets a weak value that has become inaccessible, it removes the key-value pair.

```
> t := []

> setmetatable(t, ['__weak' ~ 'v'])

> v := [1]; insert v into t

> v := [2]; insert v into t

> environ.gc()

> t:
[2 ~ [2]]
```

Do not change the '__weak' field after it has been assigned to an object, as the behaviour would be undefined. The **insert** and **delete** statements will reject manipulation of weak tables and sequences.

6.21 Extending Built-in Functions

You may redefine existing built-in functions if you want to change their behaviour or extend its features. You can either write a completely new replacement from scratch or use the original function in your modified version. Your new procedure can then be called with the same name as the original one.

Note that only Agena functions written in C or in the language itself can be redefined, and that operators cannot.

In Agena, each mathematical function f works as follows: if a number x, which by definition represents a value in the real domain, is passed to them, then the result f(x) will also be in the real domain. If x is a complex value, then the result will be in the complex domain.

Suppose that you want to automatically switch to the complex domain if a function value in the real domain could not be determined, i.e. if f(x) = **undefined**. An example is:

```
> root(-2, 2):
undefined
```

On the interactive level enclose the new procedure definition with the **scope** and **epocs** keywords. This is necessary because on the interactive level, each statement entered at the prompt has its own scope and thus local variables cannot be accessed in the statements thereafter.

The new function definition might be:

```
> scope
>
>    # save the original function in a `hidden` variable
>    local oldroot := root;
>
```

```
>      # define the substitute
>      root := proc(x, n) is  # new definition
>         local result := oldroot(x, n);
>         if result = undefined then  # switch to complex domain
>            result := oldroot(x+0*I, n)
>         fi;
>         return result
>      end;
>
> epocs;
```

The original function root is stored to the local oldroot variable so that the user can no longer directly access it.

```
> root(-2, 2):
8.6592745707194e-017+1.4142135623731*I
```

If you wish to permanently use your redefined functions, just put them into the initialisation file, located either in the `lib` folder of your Agena installation, or your home directory. See Appendix 6 for further information.
Since files have their own `scope`, the **scope** and **epocs** keywords are no longer needed (but can be left in the file).

6.22 Closures: Procedures that Remember their State

A procedure can remember its state. This state is represented by the function's local variables which survive and retain their values even after the call to the procedure has finished. Such procedures are also called `closures`.

So with successive calls, the procedure can access these values again and re-use them.

Let us define an iterator function that returns an element of a table one after the other:

```
> traverse := proc(o :: table) is
>    local count := 0;
>    return proc() is
>       inc count;
>       return o[count]
>    end
> end;
```

The `traverse` procedure is called a `factory` for it creates and returns the closure which we assign to the name `iterator`:

```
> tbl :=['a', 'b', 'c'];

> iterator := traverse(tbl);
```

The `iterator` function remembers its state and can be called like `normal` functions:

```
> iterator():
a
```

What happened ? The call to `traverse` with the table `tbl = ['a', 'b', 'c']` as its only argument initialised the variable `count` and assigned it to `0`. The table you passed is also stored to the closure's internal state since technically, parameters are local variables. With the first call to `iterate`, `count` was incremented from `0` to `1`, so that the first element of the table, i.e. `tbl[1]`, could be returned thereafter.

```
> iterator():
b

> iterator():
c
```

Since the table now has no more elements left (count = 4), the iterator now returns **null**, since `tbl[4]` = **null**.

```
> iterator():
null
```

You can define more than one closure with a factory at the same time, each being completely independent from the others:

```
> iterator2 := traverse(['a', 'b', 'c']);

> iterator2():
a

> iterator2():
b

> iterator3 := traverse(['a', 'b', 'c']);

> iterator3():
a
```

In Chapter 5, we have already introduced **for/in** loops that can iterate over functions. There are various ways to accomplish this.

In general, one or two loop control variables are given to the left of the **in** keyword, followed by the function and up to two further variables to its right.

Example 1: With function **next**, iterate table `tbl` and pass **null** as the initialiser to get its first entry. The respective values in `tbl` are assigned to loop control variable `i`:

```
> tbl := [10, 20, 30, 'a' ~ 40];

> for i in next, tbl, null do  # equivalent to `for i in tbl do`
>    print(i)
> od;
10
20
30
40
```

Example 2: Same as Example 1 but with two control variables `k`, `v` storing the respective table key and value, in this order.

```
> for k, v in next, tbl, null do
>    print(k, v)
> od;
1       10
2       20
3       30
a       40
```

Example 3: Retrieve only the table keys.

```
> for keys k in next, tbl, null do
>    print(k)
> od;
1
2
3
a
```

for/in loops iterate over factories, as well. Just some examples:

```
> gmatch := proc(s) is
>    local c, p := 0, strings.gmatch(s, '%a+');  # p is assigned a factory
>    return proc() is
>       local word := p();
>       return when word = null with null;
>       inc c;
>       return c, word  # return position and word
>    end
> end;

> s := 'hello world from Agena';

> f := gmatch(s);

> for i in f do
>    print(i)
> od;
hello
world
from
Agena

> f := gmatch(s);

> for k, v in f do
>    print(k, v)
> od;
1       hello
2       world
3       from
4       Agena

> f := gmatch(s);

> for keys k in f do
>    print(k)
> od;
1
2
```

```
3
4
```

6.23 Self-defined Binary Operators

A procedure `f` of two arguments `x, y`

```
> plus := proc(x, y) is return x + y end;
```

can be called like a binary operator through the syntax `x f y`:

```
> 1 plus 2:
3
```

When using a function as a binary operator, it has always the highest precedence.

6.24 OOP-style Methods on Tables

Agena supports OOP-style methods. For a table object representing a bank account,

```
> account := ['balance' ~ 0];
```

define the following method (please note the two `@` tokens):

```
> proc account@@deposit(x) is
>    inc self.balance, x;
>    return self.balance
> end;
```

The name `self` always refers to the table object, here `account`. Call the method using two `@` characters:

```
> account@@deposit(100)
```

Query the object.

```
> account:
[balance ~ 100, deposit ~ procedure(016D6820)]
```

Let us define a method for withdrawing an amount of money. Instead of the **proc** statement, we will now use the standard := assignment:

```
> account@@withdraw := proc(x) is
>    if x < 0 then error('Error, value must be non-negative.') fi;
>    dec self.balance, x;
>    return self.balance
> end;
```

To set up new accounts that inherit the methods and characteristics associated with the `account` object, assign the metatable of the `account` object to the freshly created account using the **setmetatable** function, and force Agena to search for the methods or its balance stored to `account` by proper indexing (i.e. `self.__index`

:= `self`). Thus, we use the `account` object as a prototype inherited by individual accounts. To explore the metatable of an object, call **getmetatable**.

```
> proc account@@new(o) is
>    o := o or [];          # if not given, create object with its initial
>                           # balance taken from the current state of `account`
>    setmetatable(o, self); # assign metatable of `account` object
>                           # (i.e. `self`) to new table
>    self.__index := self;  # inherit methods from `account` object
>    return o
> end;

> a := account@@new();

> a.balance:
100
```

Set up a new account with its initial balance set to zero:

```
> b := account@@new(['balance' ~ 0]);
```

Pay into the bank 200 currency units.

```
> b@@deposit(200):
200
```

If you want to create a different class of accounts, e.g. accounts on credit that own all the features of `account` but do not allow any overdraft, just assign an `account` object to it by calling the `new` method (do not just assign `account` to `creditaccount`):

```
> creditaccount := account@@new();
```

and overwrite the `withdraw` method:

```
> proc creditaccount@@withdraw(x :: number) is
>    if x < 0 then error('Error, value must be non-negative.') fi;
>    if x > self.balance then error('Error, not enough credit.') fi;
>    dec self.balance, x;
>    return self.balance
> end;

> c := creditaccount@@new();

> c@@withdraw(1000):
Error, not enough credit.
```

Since `b` is an unlimited account, we can withdraw money as much as we want, as its `withdraw` metamethod has not been replaced.

```
> b@@withdraw(1000):
-800
```

6.25 Assigning Tables to Procedures

As an alternative to storing values into the registry (see Chapter 6.31) or using closures (Chapter 6.22), you can assign a table to a procedure with the **store**

feature. The table will remain active during the entire Agena session and you can read from or write values to it in subsequent calls to the function.

This feature is thrice as fast as interacting with the registry, but only half as fast as closures. The table can be accessed through the **store** keyword which can also be indexed:

```
> f := proc() is
>    feature store;
>    store[1] := Pi;
>    store.count := (store.count or 0) + 1;
>    return store, store[1], store.count
> end;

> f()
[3.1415926535898, 1] 3.1415926535898 1
```

To get access to the internal store, call **debug.getstore** which returns its reference. You can both inspect this table as well as inject values into the store. In the following example we define a sine function with precomputed coefficients:

```
> zxsine := proc(x :: number) is  # ZX Spectrum SIN emulation
>    feature store;  # activate the internal store
>    local w, z;
>    x *:= 0.5/Pi;
>    x -:= entier(x + 0.5);
>    w := 4*x;
>    if w > 1 then
>       w := 2 - w
>    elif w < -1 then
>       w := -w - 2
>    fi;
>    z := 2*w*w - 1;
>    return w*(store[1] + z*(store[2] + z*(store[3] + z*(store[4] +
>       z*(store[5] + z*store[6]))))))
> end;

> _coeffs := // 1.267162131 -0.284851843 0.18226552e-1 -0.546208e-3
>             0.9480e-5 -0.112e-6 \\;
```

To get a reference to the store, execute:

```
> _store := debug.getstore(zxsine);
```

Insert coefficients to the store,

```
> for i in _coeffs do
>    insert i into _store
> od;
```

and do some cleanup thereafter:

```
> _store, _coeffs -> null;
```

Voila:

```
> zxsine(Pi/4):
0.70710678125
```

Of course, you can mix **store** tables with remember tables. For another example, see Chapter 6.31.

6.26 Summary on Procedures

The following diagram tries to summarise all features of a procedure.

Procedure

```
proc(v::type, ?) :: type is
   local r;
   global _Eps;
   r := v;
   for i in varargs do
      inc r, i + _Eps
   od;
   return r
end
```

Input

No Parameters
Parameters
Variable Parameters

Type Checks

Arguments
Return

Output

No Return
One Return
Multiple Returns

Remember Table

Read/Write Table
Read-Only Table

Domains

State (Closure)
Storage Table
Scope
Environment

Procedure Types

Standard Agena Type
User-Defined Type

Forms

Multi-Line Procedures
One-Line Functions

6.27 I/O

Agena features various functions to deal with files, to read lines and write values to them. Keyboard interaction is supported, too, as is interaction with other applications. Most of the functions have been taken from Lua. All the functions for input/output are included in the **io** and the **binio** packages.

Read and write access to files usually is conducted through file handles. At first, a file is opened for read or write operations with the **io.open** function. Then you apply the respective read or write functions and finally close the file again by calling **io.close**.

6.27.1 Reading Text Files

Open a file and store the file handle to the name `fh`:

```
> fh := io.open('d:/agena/src/change.log'):
file(7803A6F0)
```

Read the first ten characters:

```
> io.read(fh, 10):
Change Log
```

Read the next ten characters:

```
> io.read(fh, 10):
 for Agena
```

Close the file:

```
> io.close(fh):
true
```

Besides file handles, many I/O functions also accept file names. For example, the **io.lines** procedure reads in a text file line by line. It is usually used in **for** loops. The respective line read is stored to the loop key, the loop value is always **null**. The function opens and closes the file automatically.

```
> for i, j in io.lines('d:/agena/lib/agena.ini') do
>    print(i, j)
> od
execute := os.execute;    null
getmeta := getmetatable;        null
setmeta := setmetatable;        null
```

6.27.2 Writing Text Files

To write numbers or strings into a file, we must first create the file with the **io.open** function. The second argument `'w'` tells Agena to open it in `write` mode.

```
> fh := io.open('d:/file.txt', 'w');
```

As mentioned above, **io.open** returns a file handle to be used in subsequent I/O operations.

```
> io.write(fh, 'I am a text.');
```

If you would like to include a newline, pass the `'\n'` string,

```
> io.write(fh, 'Me ', 'too.', '\n');
```

or use the **io.writeline** function which automatically adds a newline to the end of the input. The next statement writes the number π to the file.

```
> io.writeline(fh, Pi);
```

After all values have been written, the file must be closed with **io.close**.

```
> io.close(fh);
```

The statements presented above produce the file contents:

```
I am a text.Me too.
3.1415926535898
```

We can append text to a file we have already created. In order to append - and not to overwrite existing - text, use the `'a'` switch in the call to **io.open**[17]. Using the `'w'` switch would replace the text already existing with the new one. See Chapter 12.1 for further options accepted by **io.open**.

Tables, sets or sequences cannot be written directly to files, they must be iterated using loops so that their keys and values - which must be numbers, booleans or strings - can be stored separately to the file thereafter. The same applies to pairs: use the **left** and **right** operators to write their components.

The following statements write all keys and values of a table to a file. The keys and values are separated by a pipe '|', and a newline is inserted right after each key~value pair. Note that you can mix numbers and strings.

```
> a := [10, 20, 30];

> file := io.open('d:/table.text', 'w');

> for i, j in a do
>    io.write(file, i, '|', j, '\n')
> od;

> io.close(file);
```

Hint: To create UNIX text files on DOS-like systems, such as DOS, OS/2, Windows, just open the text file in binary mode, e.g. `io.open('d:/table.text', 'wb')`. This avoids carriage return control codes to be added to the file with each line break.

See Chapter 12.1 for a description of all **io** package functions.

If you have trouble with character encoding, the converters **strings.tolatin**, **strings.toutf8**, **strings.diamap** or the aconv package might help you.

6.27.3 Keyboard Interaction

The **io.read** function allows to enter values interactively via the keyboard when called with no argument. Use the RETURN key to complete the input. The value returned by **io.read** is a string. If you would like to enter and process numbers thereafter, use the **tonumber** function to transform the string into a number.

```
> a := io.read();
10
```

---

[17] See Chapter 12.1 for further options accepted by **io.open**.

```
> a:
10

> type(a):
string

> tonumber(a)^2:
100
```

All available keyboard functions are:

Procedure	Details
io.anykey	Checks whether a key has been pressed and returns **true** or **false**.
io.getkey	Waits until a key is pressed and returns its ASCII value. This function is not available on all platforms.
io.read	If called with no arguments, reads one or more characters from the keyboard until the RETURN key is being pressed. The return is a string.

Table 19: Functions to read the keyboard

6.27.4 Default Input, Output, and Error Streams

Agena provides aliases to the standard input, output, and error channels known from C:

*   **io.stdin**, the standard input stream, used to input data, usually the keyboard,
*   **io.stdout**, the standard output stream, used to output data, usually the console,
*   **io.stderr**, the standard error stream, used for error messages and diagnostics, usually the console.

Examples:

```
> io.writeline(io.stdout, 'Okay');
Okay

> io.writeline(io.stderr, 'Not okay');
not okay
```

6.27.5 Locking Files

Agena allows files to be locked so that only the current process can read or write data to them. This feature prevents corruption to files during write operations or reading invalid data when other programmes try to access them. See **io.lock** and **io.unlock** in Chapter 12.1 for further information.

6.27.6 Interaction with Applications

You can call another application, pass data to it and receive data from the application with the **io.popen** function. The function returns a file handle, so that you can receive the information returned (from the stdout channel of the called programme) for further processing.

To get a listing of all files in the current directory, enter:

```
> p := io.popen('ls'):
file(77602960)

> io.readlines(p):
[ads.c, agena.c, etc.]
```

Finally, close the connection.

```
> io.close(p)
```

If you pass the `'w'` option to **io.popen** as a second argument, you can send further data to the external programme:

```
> p := io.popen('cat', 'w')

> io.write(p, 'Hello ')

> io.write(p, 'World\n')

> io.close(p)
Hello World
```

If you want to receive data from the stderr channel, or suppress output at the Agena console, include the respective redirection instruction, which may vary among operating systems, in the first argument to **io.popen**.

6.26.7 CSV Files

Comma-separated value files can be read and written conveniently by **utils.readcsv** and **utils.writecsv**. This function provides various options to further process the data being read. See Chapter 16.1 for further details.

6.27.8 XML Files

XML files are imported and converted to Agena data structures with **utils.readxml** or **xml.readxml**. XML files can be created with **utils.encodexml** and **io.write**. Chapter 16.1 and 12.5 offers further information on how to do this.

6.27.9 dBASE III/IV Files

The xbase package can read and write dBASE III/IV-compatible files. See Chapter 12.3 for details.

6.27.10 INI Files

The **utils.readini** and **utils.writeini** functions deal with traditional INI initialisation files.

6.28 Linked Lists

With large tables, sometimes it may be very costly to insert or delete an element with the **put** and **purge** functions because all elements after the insert or deletion position must either be shifted up- or downwards. This is also true with sequences and registers.

In addition, iterating a table with the **for/in** statement does not ensure that the keys are traversed in ascending order[18].

In these cases you may use the **llist** package implementing linked lists which store elements in a sequential order and where each value also links to its successor (and predecessor). Just take a look at the examples at the end of this subchapter.

The benefit of using linked list in these situations is a speed increase of at least 600 %, but may be very much larger.

To see how a linked list works, let us create one manually. First, establish a root which indicates the end of the list.

```
> list := null;
```

Now we insert the numbers -2, -1 and 0 into this list, so that it contains the elements 0, -1, -2, in this order.

```
> list := ['data' ~ -2, 'next' ~ list];

> list := ['data' ~ -1, 'next' ~ list];

> list := ['data' ~ 0, 'next' ~ list];
```

To traverse the list, we use a new reference so that the original list is not changed:

```
> l := list;

> while l do
>    print(l.data)
>    l := l.next
> od;
0
-1
-2
```

To insert an element somewhere in the list, we enter:

```
> l := list;

> while l do
>    if l.data = -1 then
>        l.next := ['data' ~ -1.5, 'next' ~ l.next];
>        break
>    fi;
>    l := l.next
> od;
```

---

[18] See **skycrane.iterate**.

```
> l := list;

> while l do
>    print(l.data)
>    l := l.next
> od;

0
-1
-1.5
-2
```

It may often be useful to add further information to a linked list to save unnecessary traversal, e.g. the position of the element or the predecessor.

Instead of implementing singly- or doubly-linked lists yourself, use the **llist** package. First initialise it,

```
> import llist
```

and create an empty list.

```
> L := llist.list():
llist()
```

Now add 0 to it

```
> llist.append(L, 0);
```

and also put -2 to its beginning.

```
> llist.prepend(L, -2);

> L:
llist(-2, 0)
```

Insert -1 at position 2. As you see, the original element at this position is not deleted but shifted to open space.

```
> llist.put(L, 2, -1):

> L:
llist(-2, -1, 0)
```

To delete an element at a position, enter:

```
> llist.purge(L, 2):

> L:
llist(-2, 0)
```

The **size** operator determines the number of all elements in a linked list.

```
> size L:
2
```

To determine a specific element, index it as usual:

```
> L[1]
-2
```

Passing an index that does not exist, simply results to **null**.

Finally, to replace an element, use a usual assignment statement.

```
> L[2] := -1

> L:
llist(-2, -1)
```

You may have a look at unrolled singly-linked lists, which are also provided by the **llist** package for high-speed processing. The ulist functions have the same name as those for llists, and almost the same syntax, so here is just a small example:

```
> import llist            # _not_ `import ulist`

> a := ulist.list(64)     # 64 slots per node

> for i to 11 do ulist.append(a, i) od  # fill ulist with numbers 1 to 11

> ulist.put(a, 5, 100);   # insert 100 at position 5

> a := ulist.dump(a);     # convert ulist into a sequence and dump it
>                         # from memory

> print(a)
seq(1, 2, 3, 4, 100, 6, 7, 8, 9, 10, 11)
```

Finally, functions to work on doubly-linked lists are available in table **dlist**. Read and write access to elements in doubly-linked lists is around twice as fast as for singly-linked lists:

```
> l := dlist.list('Algol 68', 'Maple', 'Lua', 'SQL');

> dlist.append(l, 'Agena');  # add new entry to the end of the list

> l[-1]:
Agena

> dlist.prepend(l, 'Agena'); # add new entry to the start of the list

> l[1]:
Agena

> dlist.purge(l, 1);          # delete first entry

> l[1]:
Algol 68

> dlist.purge(l, -1);  # delete last entry

> l[-1]:
SQL
```

```
> # insert a new value into the middle of the list, shifting elements into
> # open space
> dlist.put(l, 3, 'Agena');

> dlist.toseq(l):
seq(Algol 68, Maple, Agena, Lua, SQL)

> f := dlist.iterate(l);  # iterate through the list

> f():
Algol 68
(etc.)
```

6.29 Numeric C Arrays

Agena numbers can alternatively be processed using numeric C arrays. The numarray package supports C doubles, signed 4-byte integers (int32_t), and unsigned chars. See Chapter 10.6 for further details.

While C numeric arrays consume less memory than Agena's built-in structures, operations are slower.

6.30 Userdata and Ligthuserdata

Some Agena packages such as linked lists and numarrays implement data structures by so-called userdata, i.e. C structures that are garbage-collected by the interpreter provided that a '__gc' metamethod exists.

Likewise, lightuserdata are pointers to any C objects but programmers writing C libraries have to implement their own garbage collection procedures.

To the ordinary programmer writing code exclusively in the Agena language, userdata and lightuserdata are irrelevant as this kind of data can only be accessed through functions written in C.

6.31 The Registry

The registry is an interface between Agena and its C virtual machine which mainly stores values needed by userdata, metatables of libraries written in C, open files, and loaded libraries. It can also be used to exchange data between the C environment and Agena, or between Agena functions in general. See Chapter 6.25 for a faster alternative if you know that a function does not need to exchange data with other functions.

debug.getregistry gives full access to the registry but should be used carefully. It is recommended to revert to the functions of the **registry** package to read, add or delete registry data or to modify C library metatables, and to exclude the **debug** library from sandboxes (see Chapters 6.15, 7.40 and 7.53).

Registry entries indexed by integral keys refer to data occupied by userdata objects, which for example are used by the **llist** and **numarray** libraries. The **registry** library, however, does not expose these values to Agena.

Following is an example how you can use this feature:

```
> watch := proc(x) is
>    local id, t, val;
>    t := time();
>    # create light userdata as registry key
>    id := 'baselib_watch';
>    unassigned registry.get(id) ? registry.anchor(id, 0);
>    if x then  # any argument given ? -> initialise / reset the clock
>        registry.anchor(id, 0);
>        return
>    fi;
>    val := registry.get(id);  # get old time (in seconds)
>    if val = 0 then  # start clock
>        registry.anchor(id, t);  # assign a new value to registry
>        t := 0
>    else  # return elapsed time and set clock to current time
>        t -:= val;
>        registry.anchor(id, time())
>    fi;
>    return t
> end;
```

In comparison, an implementation using an internal store table would be:

```
> watch := proc(x) is
>    feature store;
>    local id, t, val;
>    val := store[1];  # get old time (in seconds)
>    unassigned val ? store[1] := 0; # initialise with the first call
>    t := time();
>    if x then  # reset the clock but do not turn it on again
>        store[1] := 0;
>        return
>    fi;
>    if val = 0 then  # start clock
>        store[1] := t;
>        t := 0
>    else  # return elapsed time and set clock to current time
>        t -:= val;
>        store[1] := time()
>    fi;
>    return t
> end;
```

Part **Two**

Reference

Chapter **Seven**

The Libraries

7 The Libraries

The standard libraries taken from the Lua 5.1distribution provide useful functions that are implemented directly through the C API. Some of these functions provide essential services to the language (e.g., **next** and **getmetatable**; others provide access to `outside` services (e.g., I/O); and others could be implemented in Agena itself, but are quite useful or have critical performance requirements that deserve an implementation in C (e.g., **sort**) .

The following text is based on Chapter 5 of the Lua 5.1 manual and includes all the new operators, functions, and packages provided by Agena.

Lua functions which were deleted from the code are not described. References to Lua were not deleted from the original text. If an explanation mentions Lua, then the description also applies to Agena.

All libraries are implemented through the official C API and are provided as separate C modules. Currently, Agena has the following standard libraries:

- the basic library,
- package library,
- string library,
- table library,
- mathematical library,
- two input and output libraries,
- operating system library,
- environmental libraries,
- debug facilities.

Except for the basic and the package libraries, each library provides all its functions as fields of a global table or as methods of its objects. Agena operators have been built into the kernel (the Virtual Machine), so they are not part of any library.

Chapter **Eight**

Basics

8 Basics

The basic library provides some core functions to Agena. If you do not include this library in your application, you should check carefully whether you need to provide implementations for some of its facilities.

For logical operators, please see Chapter 4.8.

Summary of functions:

Checks

> **$$**, **abs**, **alternate**, **assigned**, **assume**, **binsearch**, **filled**, **has**, **isequal**, **rawequal**, **recurse**, **satisfy**, **whereis**.

Extraction

> **$**, **bottom**, **columns**, **descend**, **duplicates**, **getentry**, **left**, **max**, **min**, **next**, **ops**, **rawget**, **recurse**, **right**, **top**, **unique**, **unpack**, **values**.

Types

> **checkoptions**, **checktype**, **float**, **gettype**, **isboolean**, **iscomplex**, **isint**, **isnegative**, **isnegint**, **isnonnegint**, **isnonposint**, **isnumber**, **isnumeric**, **ispair**, **isposint**, **ispositive**, **isseq**, **isstring**, **isstructure**, **istable**, **nan**, **nonneg**, **optboolean**, **optcomplex**, **optint**, **optnonnegative**, **optnonnegint**, **optnumber**, **optposint**, **optpositive**, **optstring**, **settype**, **type**, **typeof**.

Counting

> **countitems**, **size**.

Data Manipulation

> **@**, **augment**, **getbit**, **map**, **move**, **prepend**, **purge**, **put**, **rawset**, **reduce**, **remove**, **select**, **selectremove**, **setbit**, **sort**, **sorted**, **subs**, **toreg**, **toseq**, **toset**, **totable**, **zip**.

Data Generation

> **iterate**, **tables.new**, **sequences.new**, **registers.new**.

Error Handling

> **argerror**, **error**, **protect**, **xpcall**.

Libraries

> **readlib**, **with**.

Files

> **read**, **save**.

Output

> **print**, **printf**, **write**, **writeline**.

Parsing

> **load**, **loadfile**, **loadstring**.

Cantor Operations

> **bintersect**, **bisequal**, **bminus**.

Metatables

> **getmetatable**, **setmetatable**.

Miscellaneous

> **bye**, **clear**, **restart**, **time**.

`f @ obj`

`f @ g`

In the first form, the operator maps a function `f` to all the values in table, set, sequence, register, string or pair `obj`. `f` should be a univariate function and return only one value. The type of return is the same as of `obj`. If `obj` has metamethods or user-defined types, the return will also have them.

If `obj` is a string, `f` is applied on all of its characters from the left to right. The return is a sequence of function values.

Examples:

```
> << x -> x^2 >> @ [1, 2, 3]:
[1, 4, 9]

> << x -> x > 1 >> @ [1, 2, 3]:
[false, true, true]
```

In the second form, the function creates the composition of two functions `f` @ `g` = `f`(`g`(x)) and returns it is a new function (`f` @ `g`)(x). `f` and `g` may be univariate or multivariate and also return multiple results.

Example:

```
> # first take root, then negate

> h := << x -> -x >> @ << x -> sqrt x >>

> h(2):
-1.4142135623731
```

See also: @ and **$$** operators, **map**, **reduce**, **remove**, **select**, **subs**, **times**, **zip**.

f $ obj

Returns all values in table, set, sequence or register `obj` that satisfy a condition determined by function `f`. `f` should be a univariate function and return at least one value. In the multivariate case, all results but the first are ignored.

```
> << x -> x > 1 >> $ [1, 2, 3]:
[2, 3]
```

If present, the function also copies the metatable and user-defined type of `obj` to the new structure.

Please note that if `obj` is a table, the return might include holes. With `obj` a register, all values up to the current top pointer are evaluated, and the size of the returned register is equal to the number of the elements in the return.

See also: @ operator, **countitems**, **descend**, **map**, **remove**, **selectremove, subs**, **unique**, **values**, **zip.**

f $$ obj

Checks whether at least one element in table, set, sequence or register `obj` satisfies the condition defined by function `f` and returns **true** or **false**. `f` should be a univariate function and return at least one value. In the multivariate case, all results but the first are ignored.

```
> << x -> x < 1 >> $$ [1, 2, 3]:
false
```

Please note that if `obj` is a table, the return might include holes. With `obj` a register, all values up to the current top pointer are evaluated.

See also: @ operator, **countitems**, **descend**, **map**, **remove**, **selectremove, subs**, **unique**, **values**, **zip.**

`abs (x)`

If x is a number, the **abs** operator will return the absolute value of x. With complex numbers, the magnitude $\sqrt{\text{real(x)}^2 + \text{imag(x)}^2}$ is evaluated (see also: **cabs**).

If x is a Boolean, it will return 1 for **true**, 0 for **false**, and -1 for **fail**.

If x is null, **abs** will return -2.

If x is a string of only one character, **abs** will return the ASCII value of the character as a number. If x is the empty string or longer than length 1, the function returns fail.

`alternate (x, y)`

Returns x if y evaluates to **null**, else returns y. This is equivalent to **if** y = **null then** x **else** y **fi**, which is not equal to y **or** x.

See also: **or** operator.

`argerror (x, procname, message)`

Receives any value x, the name of procedure `procname` (a string) where x did not satisfy anything, the error message text `message`, and appends the user-defined type or if not defined the basic type of x. Thus it returns the error message: 'Error in `procname`: `message`, got <type of x>.'.

The function is written in Agena and included in the lib/library.agn file.

See also: **error**.

`assigned (obj)`

This Boolean operator checks whether any value different from **null** is assigned to the expression `obj`. If `obj` is already a constant, i.e. a number, boolean including **fail**, or a string, the operator always returns **true**. If `obj` *evaluates* to a constant, the operator also returns **true**. See also: **unassigned**.

`assume (obj [, message])`

Issues an error when the value of its argument `obj` is **false** (i.e., **null** or **false**); otherwise, returns all its arguments. `message` is an error message; when absent, it defaults to 'assumption failed'.

`augment (obj1, obj2 [, ···])`

Joins two or more tables, sequences or registers `obj1`, `obj2` together horizontally. The arguments must either be tables, sequences or registers only. All structures must be of the same size. The type of return is determined my the type of the arguments.

The function is written in Agena and included in the lib/library.agn file.

See also: **columns**, **linalg.augment**.

`beta (x, y)`

Computes the Beta function. `x` and `y` are numbers or complex values. The return may be a number or complex value, even if `x` and `y` are numbers. The Beta function is defined as: $\mathrm{Beta}(x, y) = \frac{\Gamma x * \Gamma y}{\Gamma(x+y)}$, with special treatment if `x` and `y` are integers.

`binsearch (o, x [m [, l [, r]]])`

Performs a binary search for `x` in the *sorted* table, sequence or register `o`. You may optionally specify the left border `l` and the right border `r` in `o` where to search for `x`, by default `l` is 1 and `r` is size `o`. The very first element in `o` to be checked is given by `m` which by default is $(l + r) \setminus 2$.

The function returns **true** on success or **false** otherwise. The second return is the index position of the last element checked before the function returns.

You may have to sort `o` before invoking the function, otherwise the result would be incorrect.

See also: **in** operator.

`bintersect (obj1, obj2 [, option])`

Returns all values of table, sequence or register `obj1` that are also values in table or sequence `obj2`. `obj1` and `obj2` must be of the same type. The function performs a binary search in `obj2` for each value in `obj1`. If no option is given, `obj2` is sorted before starting the search. If you pass an option of any value then `obj2` should already have been sorted, for no correct results would be returned otherwise.

With larger structures, this function is much faster than the **intersect** operator.

The function is written in Agena and included in the lib/library.agn file.

See also: **bisequal**, **bminus**.

`bisequal (obj1, obj2 [, option])`

Determines whether the tables, sequences or registers `obj1` and `obj2` contain the same values. The function performs a binary searches. If no option is given (any value), `obj1` and `obj2 are` sorted before starting the search. If you pass an option of any type then `obj1` and `obj2` should already have been sorted, for no correct results would be returned otherwise.

With larger structures, this function is much faster than the = operator.

The function is written in Agena and included in the lib/library.agn file.

See also: **bintersect**, **bminus**.

bminus (obj1, obj2 [, option])

Returns all values of table, sequence or register `obj1` that are not values in table, sequence or register `obj2`. `obj1` and `obj2` must be of the same type. The function performs a binary search in `obj2` for each value in `obj1`. If no option is given, `obj2` is sorted before starting the search. If you pass the option then `obj2` should already have been sorted, for no correct results would be returned otherwise.

With larger structures, this function is much faster than the **minus** operator.

The function is written in Agena and included in the lib/library.agn file.

See also: **bintersect**, **bisequal**.

bottom (obj)

With the table array, sequence or register `obj`, the operator returns the element at index 1. If `obj` is empty, it returns **null**.

See also: **top**.

bye

Quits the Agena session. No arguments or brackets are needed. If a procedure has been assigned to the name **environ.onexit**, then this procedure is automatically run before exiting the interpreter. The function also conducts a final garbage collection fully closes the state of the interpreter before leaving. An example:

```
> environ.onexit := proc() is print('Tschüß !') end

> bye
Tschüß !
```

checkoptions (procname, obj, option [, ···] [, true])

Checks options passed to a given procedure, saving many lines of code in procedures.

Since an option such like `delimiter=';'` in a function call is actually passed as the pair `'delimiter':';'` you have to make sure that `real` pairs containing data (but not options) are not included in the call to **checkoptions**. See Chapter 6.6.

Its first argument `procname` - a string, not the function reference - is the name of the procedure in which the check takes place.

Its second argument `obj` - a table - represents the arguments to be checked.

The third to last arguments are pairs. The respective left operand (a string) will be checked whether one of the right operands of the pairs in `obj` is of the type passed as the right operand (a string or a basic type). See examples below.

The evaluation of `obj` works as follows: If an entry in `obj` is not a pair, it is not evaluated, ignored and not returned in the resulting table. But if the entry is a pair, it checks whether the left-hand side is a string, i.e. the name of an option. It then checks whether its right hand side is of the given type in anything passed to `option` or further options of type pair. By default, If an option in `obj` cannot be found in `option` or further options of type pair, an error will be issued. But if the very last argument is the Boolean value **true**, no error will be issued and the `unknown` option is part of the resulting table.

If successful, the return is a table where the respective left-hand side in `obj` is the key and the respective right-hand side in `obj` is the respective entry. Please play around with this new function, or have a look at the lib/skycrane.agn file in your local Agena installation, function **skycrane.scribe**. User-defined types are properly handled.

Thus:

```
> checkoptions('myproc', [1, 'neil':'armstrong'], neil=string):
> # 'neil' must be a string, number 1 will be skipped as not being a pair
[neil ~ armstrong]

> checkoptions('myproc', ['neil':'armstrong'], neil=boolean):
Error in `myproc`: boolean expected for neil option, got string.

> checkoptions('myproc', ['neil':'armstrong', 'james':'lovell'],
>    neil=string, true):
[james ~ lovell, neil ~ armstrong]
```

checktype (obj, main, sub)

Checks whether the structure `obj` is a table, set, pair, sequence or register, and whether it is of the type given by `main` (a string), and whether all its elements are of type `sub` (a string). It returns **true** or **false**. User-defined types are supported.

The function is written in Agena and included in the lib/library.agn file.

See also: **type**.

`cleanse (t)`

Empties a table, set or register and returns the emptied structure. With a register, sets all its places to **null** and returns the modified register. With tables, sets and sequences, the memory previously occupied can be reused by the interpreter.

`clear v1 [, v2, ···]`

Deletes the values in variables `v1`, `v2`, ···, and performs a garbage collection thereafter in order to clear the memory occupied by these values.

`columns (obj, p [, ···] [, 'structure'])`

Extracts the given columns `p` (etc.) from the two-dimensional table, sequence or register `obj`. The type of return is determined by the type of `obj` and is either a structure of structures if the option `'structure'` is given, or a multiple return of structures.

The function is written in Agena and included in the lib/library.agn file.

See also: **ops**, **select**, **unpack**, **values**, **linalg.column**, **utils.readscv**.

`copy (obj [, option])`

The function copies the entire contents of a table, set, pair or sequence `obj` into a new structure. If `obj` contains structures itself, those structures are also copied (by a `deep copying` method). Structures included more than once are properly aggregated to one single reference to save memory space. Metatables and user-defined types are copied, too.

With tables, if the `'array'` option is given, then the function returns just the array part of `obj`. Likewise, the `'hash'` option only extracts the hash part of `obj`.

If the option is `'nometa'`, then metatables and user-defined types will not be copied regardless of the data type of `obj`.

The type of return is determined by the type of `obj`.

The function also treats cycles (structures that directly or indirectly reference to themselves), correctly.

See also: **copyadd**.

`copyadd (obj [, ···])`

Copies all elements in table, set, sequence or register `obj` and any further optional arguments into a new structure and returns it. The result is of the same type as `obj`.

With tables, the array and hash parts are copied 1:1, i.e. the elements in the array part of `obj` are copied to the array part with the same keys, and the elements in the hash part of `obj` are copied to the hash part of the new table, with the same keys, too.

For performance reasons, substructures are not deep-copied.

The function may be used when in an expression there is a call to **unpack** or any other function returning multiple values, which is followed by one or more *subsequent* values, as in this situation multiple values returned by any function get truncated to the first value if it's not the last trailing expression.

Compare for example:

```
> f := proc(x, y) is
>    return math.sincos(x), y
> end;

> f(0, E):
0       2.718281828459
```

versus

```
> f := proc(x, y) is
   return unpack(copyadd([math.sincos(x)], y))
end;

f(0, E):
0       1       2.718281828459
```

The function is written in Agena and included in the lib/library.agn file.

See also: **copy**, **tables.include**, **insert** statement.

countitems (item, obj)

countitems (f, obj [, ···])

In the first form, counts the number of occurrences of an `item` in the structure (table, set, sequence or register) `obj`.

In the second form, by passing a function `f` with a Boolean relation as the first argument, all elements in the structure `obj` that satisfy the given relation are counted. If the function has more than one argument, then all arguments *except the first* are passed right after the name of the object `obj`.

The return is a number. The function may invoke metamethods.

See also: **select**, **bags** package.

```
descend (f, obj, [, ···] [, option])
```

Returns all elements in the structure `obj` (a table, set, sequence or register) that satisfy a given condition expressed by function `f`. The function can be multivariate and must return either **true** or **false**. The optional second and all further arguments of `f` may be passed as the third, etc. argument.

With tables, all the entries and keys are scanned.

With sequences and registers, only the entries (not the keys) are scanned.

The function performs a recursive descent if it detects tables, sets, registers or sequences in `obj` so that it can find elements in deeply nested structures. Pairs, however, are ignored.

If `obj` is a table and the option `skiphash`=**true** has been passed, then the function ignores all non-numeric keys and their corresponding values, i.e. ignores the hash part of a table.

The function returns a structure with its type depending on the type of `obj` with all the hits in no more than two levels, an example:

```
> s := seq(1, 2, 3, [1, 2, 3], seq(1, 2, 2, 4, {2, 4, 5}));

> descend(<< x -> x = 2 >>, s):
seq(2, [2], seq(2, 2), {2})

> # return all elements greater or equal 3

> ge := proc(x, y) is  # x greater or equal y ?
>    try
>       return x >= y
>    catch  # avoid comparisons of numbers with other data types
>       return false
>    yrt
> end;

> descend(ge, s, 3):
seq(3, [3], seq(4), {4, 5})
```

descend issues an error if `obj` is unassigned.

See also: **has**, **recurse**, **satisfy**, **select**.

```
duplicates (obj [, option])
```

Returns all the values that are stored more than once to the given table, sequence or register `obj`, and returns them in a new table, sequence or register. Each duplicate will be returned only once. If option is not given, the structure is sorted before evaluation since this is needed to determine all duplicates. The original structure is left untouched, however.

If a value of any type is given for `option`, the function assumes that the structure has been already sorted. The values in `obj` should either be strings or numbers if no `option` is given, otherwise the function will fail.

The function is written in Agena and included in the lib/library.agn file.

empty (obj)

This Boolean operator checks whether a table, set, register, sequence or string `obj` does not contain any item and returns **true** if so; otherwise it returns **false**.

See also: **filled**.

error (message [, level])

Terminates the last protected function called and returns `message` as the error message. **error** never returns.

Usually, **error** adds some information about the error position at the beginning of the message. The `level` argument specifies how to get the error position. With level 1 (the default), the error position is where the **error** function was called. Level 2 points the error to where the function that called **error** was called; and so on. Passing a level 0 avoids the addition of error position information to the message.

See also: **argerror**.

eval (···)

Just returns the value represented by its arguments. See also: **identity**.

everyth (n, k)
everyth (obj, k)

In the first form, returns the Agena equivalent `n` % `k` = 0, a Boolean.

In the second form, returns every given `k`-th element in the table, sequence or register `obj` in a new structure. The type of return is determined by the type of the first argument. With tables, only the array part is traversed.

_G

A global variable (not a function) that holds the global environment (that is, `_G._G = _G`). Agena itself does not use this variable; changing its value does not affect any environment, nor vice-versa. (Use **setfenv** to change environments.)

`filled (obj)`

This Boolean operator checks whether a table, set, register, sequence or string `obj` contains at least one item and returns **true** if so; otherwise it returns **false**.

See also: **empty**.

`getbit (x, pos)`

Checks for the bit at position `pos` ∈ [1, 32] in the integer `x`, and either returns **true** or **false**.

See also: **getbits**, **getnbits**, **setbit**, **setbits**, **setnbits**, **bytes.tobytes**, **numarray.getbit**.

`getbits (x [, any])`

Returns all 32 bits in the integer `x`, and returns a register of size 32 with values **true** or **false**. If any second argument is given, the register is filled with zeroes or ones instead of booleans.

See also: **getbit**, **getnbits**, **setbit**, **setbits**, **setnbits**, **numarray.getbit**.

`getnbits (x, pos, nbits)`

From the 32-bit integer `x`, starting from bit position `pos` from the right, retrieves `nbits` bits and returns a decimal value. `pos` should be in [1, 32].

`getentry (obj [, k₁, ···, kₙ])`

Returns the entry `obj[k₁, ···, kₙ]` from the table, sequence or register `obj` without issuing an error if one of the given indices `kᵢ` (second to last argument) does not exist. It conducts a raw access and thus does not invoke any metamethods.

If `obj[k₁, ···, kₙ]` does not exist, **null** will be returned. If only `obj` is given, it is simply returned.

See also: **..** operator, **{}** indexing, **getorset**.

`getmetatable (obj)`

If `obj` does not have a metatable, returns **null**. Otherwise, if the `obj`'s metatable has a '`__metatable`' field, returns the associated value. Otherwise, returns the metatable of the given `obj`.

See also: **setmetatable**.

getorset (obj, k₁, ···, kₙ, v)

Returns the non-**null** element at index `obj`[k_1, k_2, ..., k_n], where `obj` is a table, sequence or register. If any index position is invalid, the function returns **null**.

If `obj`[k_1, k_2, ..., k_n] = **null**, then the function assigns `obj`[k_1, k_2, ..., k_n] := `v` and returns `v`.

See also: **getentry**.

gettype (obj)

Returns the type - set with **settype** - of a function, sequence, set, pair or userdata `obj` as a string. If no user-defined type has been set, or any other data type has been passed, **null** will be returned.

See also: **settype**, **typeof**.

has (obj, x)

Checks whether the structure `obj` (a table, set, sequence, register or pair) contains element `x`. If `obj` and `x` are strings, checks whether at least one character in `obj` matches one of the characters in `x`.

With tables, all the entries are scanned. If `x` is not a number then the indices of the table are searched, too.

With sequences and registers, only the entries (not the keys) are scanned. With pairs, both the left and the right item is scanned. The function performs a deep scan so that it can find elements in deeply nested structures.

The function return **true** if `x` could be found in `obj`, and **false** otherwise. If `obj` <> `x` and if `obj` is a number, boolean, complex number, string, procedure, thread, userdata or lightuserdata, **has** returns **fail**.

See also: **descend**, **in**, **recurse**, **satisfy**.

identity (···)

Returns its arguments. See also: **eval**, **unpack**.

initialise (packagename [, false])
initialise (packagename , key1, key2, ··· [, false])

Assigns short names to package procedures such that:

```
name := packagename.name
```

The function works as follows:

- In both forms, **initialise** first tries to load and run the respective Agena package. The package may reside in a text file with file suffix `.agn`, or in a C dynamic link library with file suffix `.so` in UNIX and `.dll` in Windows, or both in a text file and in a dynamic link library. The function first tries to find the package in the current working directory and if it failed, in the path pointed to by mainlibname; if this fails, too, it traverses all paths in **libname** from left to right until it finds at least the C DLL or the Agena text file, or both. If a package consists of both the C DLL and an Agena text file, then they both must reside in the same folder.

- If the function does not find the package, an error will be returned.

- Next, **initialise** tries to find a package initialisation procedure. If a procedure named \``packagename`.init\` is present in your package then it is executed if the package has been found successfully.

- In the first form, if only the string `packagename` is given, short names to all functions residing in the global table `packagename` are created.

  If you do not want **initialise** to assign short names for certain functions, their names should be in the format `packagename`.aux.`procedurename`, e.g. math.aux.errormessage.

  Note that if `packagename.name` is not of type procedure, a short name is not created for this object.

- If you would like to display a welcome message, put it into the string `packagename`.initstring. It is displayed with an empty line before and after the text. An example:

  ```
  agenapackage.initstring := 'agenapackage v0.1 for Agena as of \
  May 23, 1949\n';
  ```

- In the second form, you may specify which short names are to be assigned by passing them as further arguments in the form of strings. Contrary to the first form, short names are also created for tables stored to table `packagename`.

  As opposed to the first version, **initialise** does not print any short names or welcome messages on screen.

- Further information regarding both forms:

  The function returns a table of all short names assigned.

  If the global environment variable **environ.withverbose** is set to **false**, no

messages are displayed on screen except in case of errors. If it is set to any other value or **null**, a list of all the short names loaded and a welcome message is printed.

If a short name has already been assigned, a warning message is printed. If a short name is protected (see table **environ.withprotected**), it cannot be overwritten by **initialise** and a proper message is displayed on screen. You can control which names are protected by modifying the contents of **environ.withprotected**.

For information on which folders are checked and how to add new directories to be searched by **initialise**, see **readlib**.

Note that **initialise** executes any statements (and thus also any assignment) included in the file `packagename`.agn.

The function is written in Agena and included in the lib/library.agn file.

If the last argument is the Boolean **false**, **initialise** does not print the assigned shortcuts at the console.

Note: the **import**/**alias** statement is an interface to the **initialise** function but does not require package names to be put into quotes. For example,

```
> initialise 'stats';
```

is equivalent to

```
> import stats alias;
```

See also: **readlib**, **run**, **register**, and **import**/**alias** statement.

ipairs (obj)

Returns three values: an iterator function, the table, sequence, register, string or userdata `obj`, and 0, so that the construction

```
    for i, v in ipairs(obj) do body od
```

will iterate over the pairs (1, `obj`[1]), (2, `obj`[2]), ⋯, up to the first integer key absent from the data structure.

If you pass userdata, for example a numarray, it must feature a metatable with an '__index' metamethod. Otherwise an error will be issued.

If there is nothing more to iterate, the iterator returns **null**s.

See **next** for the caveats of modifying the table during its traversal; and also: **pairs**, **factory.iterate**, **skycrane.iterate**.

Example:

```
> import numarray

> d := numarray.double(3)

> d[1] := Pi; d[2] := 2*Pi; d[3] := 3 *Pi;

> f := ipairs(d):
procedure(00410A30)

> idx, val := f(d, 0):      # pass 0 to start the iteration
1       3.1415926535898

> idx, val := f(d, idx):
2        6.2831853071796

> idx, val := f(d, idx):
3        9.4247779607694

> idx, val := f(d, idx):  # nothing left
null     null
```

isboolean (···)

Checks whether the given arguments are all of type **boolean** and returns **true** or **false**.

iscomplex (···)

Checks whether the given arguments are all of type **complex** and returns **true** or **false**.

isequal (obj1, obj2)

Equivalent to `obj1` = `obj2` and returns **true** or **false**.

The function is written in Agena and included in the lib/library.agn file.

isint (···)

Checks whether all of the given arguments are integers and returns **true** or **false**. If at least one of its arguments is not a number, the function returns **fail**.

See also: **finite**, **float**.

isnegative (···)

Checks whether all of its arguments are negative numbers and returns **true** or **false**. If at least one of its arguments is not a number, the function returns **fail**.

See also: **isnegint**, **isposint**, **isnonneg**, **ispositive**.

isnegint (···)

Checks whether all of the given arguments are negative integers and returns **true** or **false**. If at least one of its arguments is not a number, the function returns **fail**.

isnonneg (···)

Checks whether all of its arguments are zero or positive numbers and returns **true** or **false**. If at least one of its arguments is not a number, the function returns **fail**.

See also: **isnegint**, **isposint**, **isnegative**, **ispositive**.

isnonnegint (···)

Checks whether all of the given arguments are zeros or positive integers and returns **true** or **false**. If at least one of its arguments is not a number, the function returns **fail**.

isnonposint (···)

Checks whether all of the given arguments are zeros or negative integers and returns **true** or **false**. If at least one of its arguments is not a number, the function returns **fail**.

isnumber (···)

Checks whether the given arguments are all of type **number** and returns **true** or **false**.

isnumeric (···)

Checks whether the given arguments are all of type **number** or of type **complex** and returns **true** or **false**.

ispair (···)

Checks whether the given arguments are all type **pair** and returns **true** or **false**.

isposint (···)

Checks whether all of its arguments are positive integers and returns **true** or **false**. If at least one of its arguments is not a number, the function returns **fail**.

See also: **isnonposint**.

`ispositive (···)`

Checks whether all of its arguments are positive numbers and returns **true** or **false**. If at least one of its arguments is not a number, the function returns **fail**.

See also: i**snonposint**, **isposint**, **isnegative**, **isnonneg**.

`isreg (···)`

Checks whether all of its arguments are of type **register** and returns **true** or **false**.

`isseq (···)`

Checks whether all of its arguments are of type **sequence** and returns **true** or **false**.

`isstring (···)`

Checks whether all of its arguments are of type **string** and returns **true** or **false**.

`isstructure (···)`

Checks whether all of its arguments are of type **table**, **set**, **sequence** or **pair** and returns **true** or **false**.

`istable (···)`

Checks whether all of its arguments are of type **table** and returns **true** or **false**.

`left (obj)`

With the pair `obj`, the operator returns its left operand. This is equals to `obj[1]`.

See also: **right**.

`load (f [, chunkname])`

Loads a chunk using function `f` to get its pieces. Each call to `f` must return a string that concatenates with previous results. A return of **null** (or no value) signals the end of the chunk.

If there are no errors, returns the compiled chunk as a function; otherwise, returns **null** plus the error message. The environment of the returned function is the global environment.

`chunkname` is used as the chunk name for error messages and debug information.

`loadfile ([filename])`

Similar to **load**, but gets the chunk from file `filename` or from standard input, if no file name is given.

`loadstring (s [, chunkname])`

Similar to **load**, but gets the chunk from the given string `s`. To load and run a given string, use the idiom

```
assume(loadstring(s))(···)
```

Examples:

```
> f := loadstring('a := exp(1)');

> f():

> a:
2.718281828459

> f := loadstring('local x := exp(1); return x')();

> f:
2.718281828459

> f := loadstring('return exp(1)')():
2.718281828459
```

See also: **strings.dump**.

`map (f, obj [, ···] [, true])`

`map (f, g)`

In the first form, the **map** function maps a function `f` to all the values in table, set, sequence, register, string or pair `obj`. `f` must return only one value. The type of return is the same as of `obj`. If `obj` has metamethods or user-defined types, the return will also have them.

If `obj` is a string, `f` is applied on all of its characters from the left to right. The return is a sequence of function values.

If function `f` has only one argument, then only the function and the structure `obj` must be passed to **map**. If the function has more than one argument, then all arguments *except the first* are passed right after the name of the table or set.

Examples:

```
> map( << x -> x^2 >>, [1, 2, 3] ):
[1, 4, 9]

> map( << (x, y) -> x > y >>, [-1, 0, 1], 0 ):  # 0 for y
[false, false, true]
```

If the very last argument is the Boolean **true**, then the operation will be done in-place, modifying the original structure, but saving memory. After completion, the function returns the modified structure.

In the second form, the function creates the composition of two functions `f @ g = f(g(x))` and returns it is a new function (`f @ g`)(x). `f` and `g` may be univariate or multivariate and also return multiple results.

Example:

```
> # first take root, then negate

> h := map(<< x -> -x >>, << x -> sqrt x >>) # which is equivalent to:

> h := << x -> -x >> @ << x -> sqrt x >>      # which results to:

> h(2):
-1.4142135623731
```

See also: **@** operator, **iterate**, **sequences.new**, **registers.new**, **pipeline**, **reduce**, **remove**, **select**, **subs**, **times**, **zip**.

max (obj [, 'sorted'])

max (x, y)

In the first form, returns the maximum of all numeric values in table, set, sequence or register `obj`. If the option `'sorted'` is passed than the function assumes that all values in `obj` are sorted in ascending order and returns the last entry. The function in general returns **null** if it receives an empty table or sequence.

In the second form, the function returns the largest of the two numbers `x` and `y`.

See also: **min**, **math.max**, **stats.minmax**.

min (obj [, 'sorted'])

min (x, y)

In the first form, returns the minimum of all numeric values in table, set, sequence or register `obj`. If the option `'sorted'` is passed than the function assumes that all values in `obj` are sorted in ascending order and returns the first entry. The function in general returns **null** if it receives an empty table or sequence.

In the second form, the function returns the smallest of the two numbers `x` and `y`.

See also: **max**, **math.min**, **stats.minmax**.

move (obj1, start, stop, newidx [, obj2])

Copies elements from the table, sequence, register or userdata `obj1` to table, sequence, register or userdata `obj2`, performing the equivalent to the following multiple assignment: `obj2[newidx]`, ⋯ = `obj1[start]`, ⋯, `obj1[stop]`. The default for `obj2` is `obj1`, i.e. elements are shifted in the same structure. The destination range can overlap with the source range. `obj1` and `obj2` must be of the same type.

Returns the destination structure `obj2`.

Example: The following statement copies four elements in table a from position 3 up to and including 6 to a new table b, starting with index 1:

```
> a := seq('a', 'b', 'c', 'd', 'e', 'f', 'g', 'h');

> b := move(a, 3, 6, 1, seq());

> b:
seq(c, d, e, f)
```

The next statement copies four elements in a to its beginning:

```
> move(a, 3, 6, 1);

> a:
seq(c, d, e, f, e, f, g, h)
```

The function is implemented in the Agena language and is included in the lib/library.agn file.

See also: **purge**, **shift**, **swap**, **tables.move**.

mulup (obj)

Multiplies all numeric values in table, sequence or register `obj`, using round-off error correction. The return is a number. If `obj` is empty or consists entirely of non-numbers, **null** will be returned. If the structure contains numbers and other objects, only the numbers are multiplied. In tables, numeric entries with non-numeric keys are processed, as well.

See also: **sumup**, **calc.fprod**.

next (obj [, index [, sentinel]])
Allows a programme to traverse all fields of a table or all items of a set, sequence or register `obj`. With strings, it iterates all its characters. Its first argument is a table, set, string or sequence and its second argument is an index in the structure.

With tables, sequences or registers, **next** returns the next index of the structure and its associated value. When called with **null** as its second argument, **next** returns an initial index and its associated value. When called with the last index, or with **null** in an empty structure, **next** returns **null**.

With sets, **next** returns the next item of the set twice. When called with **null** as its second argument, **next** returns the initial item twice. When called with the last index, or with **null** in an empty set, **next** returns **null**.

With strings, **next** returns the position of the respective character (a positive integer) and the character. When called with **null** as its second argument, **next** returns the first character. When called with the last index, **next** returns **null**.

If the second argument is absent, then it is interpreted as **null**. In particular, you can use `next(t)` to check whether a table or set is empty. However, it is recommended to use the **filled** operator for this purpose.

If the third optional argument `sentinel` is given, and if **next** during traversal encounters an element that equals this `sentinel`, the function just returns **null**, and you may start iterating the structure again from its beginning.

With tables, the order in which the indices are enumerated is not specified, *even for numeric indices.* The same applies to set items.

The behaviour of **next** is undefined if, during the traversal, you assign any value to a nonexistent field in the structure. With tables, you may however modify existing fields. In particular, you may clear existing table fields.

See also: **factory.iterate**, **factory.cycle**, **skycrane.iterate**.

ops (index, ···)

ops (s, ···)

In the first form, if `index` is a number, returns all arguments after argument number `index`. Otherwise, `index` must be the string `'#'`, and **ops** returns the total number of extra arguments it received. The function is useful for accessing multiple returns (e.g. `ops(n, ?)`).

In the second form, the index positions (integers) in sequence `s` specify the values to be returned after the first argument to **ops**.

Example:

```
> f := << () -> 10, 20, 30, 40 >>

> ops(2, f()):
20         30         40
```

If you want to obtain only the element at `index`, put the call to **ops** in brackets.

```
> (ops(2, f())):
20

> ops(seq(2, 4), f()):
20      40
```

See also: **columns**, **ops**, **unpack**, **values**.

`optboolean (x, y [, idx [, procname]])`

The function checks whether x is a Boolean and in this case returns x. If x is **null**, it returns the Boolean y, otherwise the function issues an error. If the third argument `idx`, a number, is given, then the position `idx` will be returned in error messages. If the fourth argument `procname` is given, this name is printed as the function issuing the error.

`optcomplex (x, y [, idx [, procname]])`

The function checks whether x is a number or complex number and in this case returns x. If x is **null** it returns the number or complex number y, otherwise the function issues an error. If the third argument `idx`, a number, is given, then the position `idx` will be returned in error messages. If the fourth argument `procname` is given, this name is printed as the function issuing the error.

`optint (x, y [, idx [, procname]])`

The function checks whether x is an integer and in this case returns x. If x is **null** it returns the integer y, otherwise the function issues an error. If the third argument `idx`, a number, is given, then the position `idx` will be returned in error messages. If the fourth argument `procname` is given, this name is printed as the function issuing the error.

`optnonnegative (x, y [, idx [, procname]])`

The function checks whether x is a non-negative number and in this case returns x. If x is **null** it returns the non-negative number y, otherwise the function issues an error. If the third argument `idx`, a number, is given, then the position `idx` will be returned in error messages. If the fourth argument `procname` is given, this name is printed as the function issuing the error.

See also: **optpositive**, **optnumber**.

`optnonnegint (x, y [, procname])`

The function checks whether x is a non-negative integer and in this case returns x. If x is **null** it returns the non-negative integer y, otherwise the function issues an error. If the third argument `idx`, a number, is given, then the position `idx` will be returned in error messages. If the fourth argument `procname` is given, this name is printed as the function issuing the error.

See also: **optint**, **optposint**.

`optnumber (x, y [, idx [, procname]])`

The function checks whether x is a number and in this case returns x. If x is **null** it returns the number y, otherwise the function issues an error. If the third argument `idx`, a number, is given, then the position `idx` will be returned in error messages. If

the fourth argument `procname` is given, this name is printed as the function issuing the error.

See also: **optpositive**, **optnonnegative**.

`optposint (x, y [, idx [, procname]])`

The function checks whether `x` is a positive integer and in this case returns `x`. If `x` is **null** it returns the positive integer `y`, otherwise the function issues an error. If the third argument `idx`, a number, is given, then the position `idx` will be returned in error messages. If the fourth argument `procname` is given, this name is printed as the function issuing the error.

See also: **optint**, **optnonnegint**.

`optpositive (x, y [, idx [, procname]])`

The function checks whether `x` is a positive number and in this case returns `x`. If `x` is **null** it returns the positive number `y`, otherwise the function issues an error. If the third argument `idx`, a number, is given, then the position `idx` will be returned in error messages. If the fourth argument `procname` is given, this name is printed as the function issuing the error.

See also: **optnonnegative**, **optnumber**.

`optstring (x, y [, idx [, procname]])`

The function checks whether `x` is a string and in this case returns `x`. If `x` is **null** it returns the string `y`, otherwise the function issues an error. If the third argument `idx`, a number, is given, then the position `idx` will be returned in error messages. If the fourth argument `procname` is given, this name is printed as the function issuing the error.

`pairs (obj)`

Returns three values: the **next** function, the table `obj`, and **null**, so that the construction

```
for k, v in pairs(obj) do body od
```

will iterate over all key~value pairs or values of table `obj`.

See **next** for the caveats of modifying the table during its traversal; and also: **ipairs**, **factory.iterate**, **skycrane.iterate**.

`pipeline (f [, ···], obj [, ···])`

Maps one or more functions `f`, etc. on a table, set, sequence, register or userdata `obj`, avoiding multiple internal copies of the structure if possible.

If given a userdata `obj`, the function will change its values in-place, whereas with tables, sets, sequences and registers, the original structure `obj` will not be modified.

The return is a new structure, depending on the type of `obj`. If the function has more than one argument, then all arguments except the first are passed right after the name of object `obj`.

See also: **map**, @ operator.

`prepend (x, obj)`

Prepends an object `x` to the beginning of structure `obj`, in-place. The function returns the modified structure.

The new object can always be found at index 1, all other elements have been shifted up one index into open space. With a table, its hash part is not modified.

With a register, the function automatically increases its size by one. If `obj` is a pair, returns `x : obj`.

The function is written in Agena and included in the lib/library.agn file.

See also: **put**, **insert** statement.

`print (··· [, options])`

Receives any number of arguments, and prints their values to the console, using the **tostring** function to convert them to strings. **print** is not intended for formatted output, but only as a quick way to show a value, typically for debugging. For formatted output, use **strings.format**.

In Agena, **print** also prints the *contents* of tables and nested tables to stdout if no `__tostring` metamethods are assigned to them. The same applies to sets and sequences.

If the option `'delim':<string>` is given as the last argument, then **print** separates multiple values with the given <string> delimiter, otherwise `'\t'` is used. If the option `'nonewline':true` is passed, then Agena does not print a final newline when finishing output. The `'enclose':<string>` option encloses the values in a given substring. All other types will not be enclosed. All options can be combined.

If the kernel setting `environ.kernel('longtable')` is set to **true**, then each key~value pair is printed on a separate line, and Agena halts after **environ.more** number of lines for the user to press any key for further output. Press 'q', 'Q' or the

Escape key to quit. The default for **environ.more** is 40 lines, but you may change this value in the Agena session or in the Agena initialisation file.

You may change the way **print** formats objects by changing the respective **environ.print*** functions in the lib/library.agn file. See Appendix A5 for further details.

See also: **printf**, **io.write**, **io.writeline**, **skycrane.scribe**, **skycrane.tee**.

printf ([fh,] template, ···)

If the first argument `fh` is not given, prints the optional arguments under the control of the template string `template` to stdout, else it writes to the open file denoted by its file handle `fh`. See **strings.format** for information on how to create the template string.

Example:

```
> printf('%-10s %3d %10.2f\n', 'Carbon', 6, 12.0107);
Carbon      6       12.01

> fh := io.open('file.txt', 'w');

> printf(fh, '%-10s %3d %10.2f\n', 'Carbon', 6, 12.0107);

> close(fh);
```

See also: **print**, **io.write**, **io.writeline**, **skycrane.scribe**, **skycrane.tee**.

protect (f, arg1, ···)

Calls function `f` with the given arguments in *protected mode.* This means that any error inside `f` is not propagated; instead, **protect** simply catches the error. Note that **protect** does not work with operators.

The function either returns all results from the call in case there have been no errors, or returns the error message as a string as the only return. In case of an error, the error message is set to the global variable **lasterror**, otherwise **lasterror** is set to **null**.

lasterror is useful for checking the results of a call to **protect** as in the following:

```
    if protect(···) = lasterror then ··· fi
```

See also: **xpcall**, **try/catch** statement.

purge (obj [, pos])
purge (obj, a, b)

In the first form, the function removes from table, sequence or register `obj` the element at position `pos`, shifting down other elements to close the space, if necessary. It returns the value of the removed element. The default value for `pos` is

n, where n is the length of the table, sequence or register, so that a call `purge(obj)` removes the last element of `obj`.

In the second form, removes all elements starting from index `a` to index `b` (inclusive), moving excess elements down to close the space; the function automatically performs a garbage collection after shifting. In the 2nd form, nothing will be returned.

Use the **delete** *element* **from** *structure* statement if you want to remove any occurrence of the table value *element* from a table or sequence. You might also consider using a linked list, a data structure which supports much faster operations when inserting or deleting elements, see **llist** package in Chapter 10.7.

Note that with tables, the function only works if the table is an array, i.e. if it has positive integral and consecutive keys only. With registers, the top pointer is reduced by the number of elements removed.

See also: **move**, **prepend**, **put**, **llist.purge**, **ulist.purge**.

put (obj, [pos,] value)

Inserts element `value` at position `pos` in table, sequence or register `obj`, shifting up other elements to open space, if necessary. The default value for `pos` is n+1, where n is the current length of the structure, so that a call `put(obj, value)` inserts `value` at the end of `obj`.

Use the **insert** *element* **into** *structure* statement if you want to add an element at the current end of a structure, for it is much faster. You might also consider using a linked list, a data structure which supports much faster operations when inserting or deleting elements, see **llist** package in Chapter 10.7.

The function returns the modified structure.

See also: **prepend**, **purge**, **llist.put**, **ulist.put**.

qsumup (obj)

Raises all numeric values in table, sequence or register `obj` to the power of 2 and sums up these powers, using a precision-saving method. The return is a number. If `obj` is empty or consists entirely of non-numbers, the operator returns **fail**. If the structure contains numbers and other objects, only the powers of the numbers are added. Numeric entries with non-numeric keys are processed, as well.

The operator uses a combination of fused multiply-add and Kahan-Babuška Summation. To improve accuracy, you may sort `obj` before.

See also: **qmdev**, **sumup**, **sort**, **sorted**.

`rawequal (obj1, obj2)`

Checks whether `obj1` is equal to `obj2`, without invoking any metamethod. Returns a Boolean.

`rawget (obj, index)`

Gets the real value of `obj[index]`, without invoking any metamethod. `obj` must be a table, set, sequence or pair; `index` may be any value.

See also: **getentry**, **rawset**.

`rawset (obj, index, value)`

`rawset (obj, value)`

In the first form, sets the real value of `obj[index]` to `value`, without invoking any metamethod. `obj` must be a table, set, register, sequence or pair, `index` any value different from **null**, and `value` any value. To delete a value from any structure, pass **null** for `value`.

In the second form, the function inserts `value` into the next free position in the given structure `obj`. `obj` can be a table, set, sequence or register.

This function returns `obj`.

See also: **rawget**.

`read (filename)`

Reads an object stored in the binary file denoted by file name `filename` and returns it. The function is written in Agena and included in the lib/library.agn file.

See also: **save**.

`readlib (packagename [, packagename2, ···] [, true])`

Loads and runs packages stored to agn text files (with filename `packagename`.agn) or binary C libraries (`packagename`.so in UNIX, `packagename`.dll in Windows), or to both.

If **true** is given as the last argument, the function prints the search path(s), and also quits and prints some diagnostics if a corrupt C library has been found.

The function first tries to find the libraries in the current working directory, and thereafter in the path in **mainlibname**. If it fails, it traverses all paths in **libname** until it finds them. If it finds a library and the current user has at least read permissions for it, it is initialised. On successful initialisation, the name of the package is entered into the **package.readlibbed** set.

Note that if a package consists both of a C DLL and an Agena text file, they should both be located in the very same folder as **readlib** does not search for them across multiple paths and may thus initialise a package only partially.

Make sure that on the operating system level the environment variable AGENAPATH has been set, that the individual paths are separated by semicolons and that they do not end in slashes. In UNIX, if AGENAPATH has not been set, **readlib** by default searches in `/usr/agena/lib`.

In OS/2 and Windows, the Agena installation programme automatically sets AGENAPATH. If it failed, or you want to modify its contents, you may manually set the variable like in the following examples, assuming that the Agena libraries are located in the `d:\agena\lib` folder and optionally in the `d:\agena\mypackage` folder.

```
SET AGENAPATH=d:/agena/lib   or
SET AGENAPATH=d:/agena/lib;d:/agena/mypackage
```

In UNIX, you may execute one of the following statements in your shell, assuming that the Agena libraries are located in the `/home/usr/agena/lib` folder and optionally in the `/home/usr/agena/mypackage` folder.

```
SET AGENAPATH=/home/usr/agena/lib   or
SET AGENAPATH=/home/usr/agena/lib;/home/usr/agena/mypackage
```

In DOS, you have to set AGENAPATH in the autoexec.bat file:

```
SET AGENAPATH=d:/agena/lib   or
SET AGENAPATH=d:/agena/lib;d:/agena/mypackage
```

Of course, packages may reside in other directories as well. Just enter further paths to **libname** as you need them.

The function returns **true** if all the packages have been successfully loaded and executed, or **fail** if an error occurred.

Hint: the **import** statement is an interface to **readlib** (and **initialise**), but does not require to put the package names into quotes. For example,

```
> readlib('stats');
```

is equivalent to

```
> import stats;
```

See also: **run**, **initialise**, **import** statement.

```
recurse (f, obj [, ···][, option])
```

Checks each element of the structure `obj` (a table, set, pair, sequence or register) by applying a function `f` on each of its elements. `f` can be a multivariate function and must return either **true** or **false**. The optional second and all further arguments of `f` may be passed as the third, etc. argument.

With tables, all the entries and keys are scanned.

With sequences and registers, only the entries (not the keys) are scanned.

The function performs a recursive descent if it detects tables, sets, pairs, registers or sequences in `obj` so that it can find elements in deeply nested structures.

If `obj` is a table and the option `skiphash`=**true** has been passed, then the function ignores all non-numeric keys and their corresponding values.

The function immediately returns **true** if the function call to any element in `obj` evaluates to **true**, and **false** otherwise. If `obj` is a number, boolean, complex number, string, **null**, procedure, thread, userdata or lightuserdata, **recurse** returns **fail**. It issues an error if `obj` is unassigned.

See also: **descend**, **has**, **satisfy**.

```
reduce (f, obj [, init [, ··· [, option]]])
```

Applies a function `f` on each item of a structure or string `obj` and returns an accumulated result.

`f` must have two or more parameters, but at least parameters x, a, where x will represent the respective item in `obj`, and a the accumulator to be updated. If `init` is given, then the accumulator is initialised with it, otherwise the accumulator is set to zero at first.

After traversal of `obj`, the accumulator will be returned. The function is equivalent to:

```
> reduce := proc(f, s, init, ?) is
>    local accumulator := init or 0;
>    for item in s do
>       accumulator := f(item, accumulator, unpack(?))
>    od;
>    return accumulator
> end;
```

For example, `reduce(<< x, a -> x + a >>, [1, 2, 3, 4])` computes the sum of the numbers in a table, i.e. 10; and `reduce(<< x, a -> a & x & '|' >>, '1234', '')` appends a pipe to each character, i.e. returns '1|2|3|4|'.

You can pass further arguments to the given accumulator function by just passing them as the fourth and following argument(s) to **reduce**. Example to compute the arithmetic mean of all the numbers in table [10, 20, 30]:

```
> tbl := [10, 20, 30];

> f := << x, a, len -> a + x/len >>

> a := reduce(f, tbl, 0, size tbl):
20
```

for:

```
> a, len := 0, size tbl;
> for x in tbl do
>    inc a, x/len
> od;
```

A counter can also be used: it can be accessed within the accumulator function by the name `_c` when passing the `_c = true` option - with `_c` starting from 1. The performance penalty, however, may be quite significant:

```
> tbl := seq(3, 3, 3);

> a := reduce(<< x, a -> a + x * 10^(_c - 1) >>, tbl, 0, _c=true):
333
```

may be up to four time slower than

```
> a := 0;

> for _c from 1 to size tbl do
>    inc a, tbl[_c] * 10^(_c - 1)
> od;
```

See also: @ operator, **map**.

_RELEASE

A global variable that holds a string containing the language name, the current interpreter main version, the subversion, and the patch level. The format of this variable is: `'AGENA >> <version>.<subversion>.<patchlevel>'`.

See also: global environment variable **environ.release**, **environ.version**.

remove (f, obj [, ··· [, newarray=true]] [, inplace=true])

Returns all values in table, set, sequence or register `obj` that do not satisfy a condition determined by function `f`, as a new table, set, sequence or register. The type of return is determined by the type of second argument, depending on the type of `obj`.

If the function has only one argument, then only the function and the table/set/register/sequence are passed to **remove**.

```
> remove(<< x -> x > 1 >>, [1, 2, 3]):
[1]
```

If the function has more than one argument, then all arguments *except the first* are passed right after the name of the table or set `obj`.

```
> remove(<< x, y -> x > y >>, [1, 2, 3], 1):   # 1 for y
[1]
```

If present, the function also copies the metatable and user-defined type of `obj` to the new structure.

Please note that if `obj` is a table, the return might include holes. If you pass the newarray=**true** option as the last argument, however, the result will be returned in a table array with consecutive positive integral keys, not preserving the original keys of the respective values determined, and not having holes; for example:

```
> remove(<< x -> x < 2 >>, [1, 2, 3]):
[2 ~ 2, 3 ~ 3]
> remove(<< x -> x < 2 >>, [1, 2, 3], newarray = true):
[2, 3]
```

With a register, all values up to the current top pointer are evaluated, and the size of the returned register is equal to the number of the elements in the return.

If the last argument is the option inplace=**true**, or the Boolean **true**, then the operation will be done in-place, modifying the original structure, but saving memory. After completion, the function returns the modified structure. (You can combine the newarray and inplace options).

See also: **cleanse**, **countitems**, **map**, **select**, **selectremove**, **subs**, **unique**, **zip**.

restart

Restarts an Agena session. No argument is needed.

If a procedure has been assigned to the name **environ.onexit**, then this procedure is automatically run before re-initialising the interpreter. An example:

```
> environ.onexit := proc() is print('Tschüß !') end

> restart
Tschüß !
```

During start-up, Agena stores all initial values, e.g. assigned package tables, in a global variable called **_origG**. Tables are copied, too, so their contents cannot be

altered in a session.

When the Agena session is reset, all values in the Agena environment are unassigned including the environment variable **_G**. The seeds used by **math.random/math.randomseed** are reset, too.

The system variables **_origG**, **libname**, **mainlibname environ.onexit**, the current working directory are not reset. **mainlibname** and **libname**, however, are reset to their original values if you issued the statement `environ.kernel(libnamereset = true)` before.

Then all entries in **_origG** are re-read and assigned to the new environment.

After this, the library base file `library.agn` and thereafter the initialisation file `agena.ini` or `.agenainit` - if present - are read and executed. Finally, restart runs a garbage collection.

`reverse (obj)`

Reverses the order of all elements in a sequence or register `obj` in-place. With tables, it reverses the elements in the array part, only. The function returns the modified structure.

See also: **strings.reverse**, **stack.reversed**.

`right (obj)`

With the pair `obj`, the operator returns its right operand. This is equals to `obj[2]`. See also: **left**.

`run (filename)`

Opens the named file and executes its contents as a chunk. When called without arguments, **run** executes the contents of the standard input (stdin). Returns all values returned by the chunk. In case of errors, **run** propagates the error to its caller (that is, **run** does not run in protected mode).

See also: **readlib**, **with**.

`satisfy (f, x [, ···] [, option])`
`satisfy (f, obj [, ···] [, option])`

In the first form, with `x` a number, complex number, string, boolean, null or userdata, calls the function `f` which should return **true** or **false**. The result is the return of this call. You may also specify optional arguments to `f`.

With `obj` a structure (second form), checks each element in `obj` by calling function f which also should return **true** or **false**. If at least one element in `obj` does not satisfy the condition checked by f, the result is **false**, and otherwise **true**.

The function performs a recursive descent if it detects tables, sets, pairs, registers or sequences in `obj` so that it can find elements in deeply nested structures. If `obj` is a table and the option `skiphash`=**true** has been passed, then the function ignores all non-numeric keys and their corresponding values.

See also: **has**, **recurse**, **descend**.

`save (obj, filename)`

Saves an object `obj` of any type into a binary file denoted by file name `filename`.

save returns an error if an object that cannot be stored to a file has been passed: threads, userdata, for example. It also returns an error if the object to be written is self-referencing (e.g. **_G**). If `obj` contains one and the same structure multiple times, e.g. n times, then **save** stores it n times.

The function locks the file when writing, avoiding file corruption if another application tries to gain access to it.

Note that **save** overwrites existing files without warning. Whereas numbers, strings, and Booleans are stored in a portable fashion so that the data can be read both on Big Endian (e.g SPARCs, PPCs) and Little Endian systems, procedures cannot.

The function is written in Agena and included in the lib/library.agn file.

See also: **read**, **io.writefile**.

`select (f, obj [, ··· [, newarray=true] [, inplace=true]])`

Returns all values in table, set, sequence or register `obj` that satisfy a condition determined by function `f`. The type of return is determined by the type of the second argument.

If `f` has only one argument, then only the function and the object are passed to **select**.

```
> select(<< x -> x > 1 >>, [1, 2, 3]):
[2, 3]
```

If the function has more than one argument, then all arguments *except the first* are passed right after the name of the object `obj`.

```
> select(<< x, y -> x > y >>, {1, 2, 3}, 1):   # 1 for y
{3, 2}
```

If present, the function also copies the metatable and user-defined type of `obj` to the new structure.

Please note that if `obj` is a table, the return might include holes. If you pass the newarray=**true** option as the last argument, however, the result will be returned in a table array with consecutive positive integral keys, not preserving the original keys of the respective values determined, and not having holes. Thus,

```
> select(<< x -> x :: number >>, ['a', 10, 20, 30, 'z'], newarray=true);
```

returns

```
[10, 20, 30]
```

instead of

```
[2 ~ 10, 3 ~ 20, 4 ~ 30]
```

If the last argument is the option inplace=**true**, or the Boolean **true**, then the operation will be done in-place, modifying the original structure, but saving memory. After completion, the function returns the modified structure. (You can combine the newarray and inplace options).

With a register, all values up to the current top pointer are evaluated, and the size of the returned register is equal to the number of the elements in the return.

See also: **cleanse**, **countitems**, **descend**, **map**, **remove**, **selectremove, subs**, **unique**, **values**, **zip.**

selectremove (f, obj [, ··· [, newarray=true]])

Combines the functionality of **select** with the one of **remove**: The first result contains all the elements of a structure `obj` (a table, set, sequence or register) that satisfy a given condition, the second result contains the elements of a structure not satisfying the condition. This may speed up computations where you need both results, maybe for post-processing, by around 33 %.

If `obj` is a table, the return might include holes. If you pass the newarray=**true** option as the last argument, however, the result will be returned in table arrays with consecutive positive integral keys, not preserving the original keys of the respective values determined, and not having holes. Examples,

```
> a := ['a', 10, 20, 30, 'z'];

> selectremove(<< x -> x :: number >>, a):
[2 ~ 10, 3 ~ 20, 4 ~ 30]        [1 ~ a, 5 ~ z];

> selectremove(<< x -> x :: number >>, a, newarray=true):
[10, 20, 30]    [a, z]
```

See also: **remove**, **select**.

`setbit (x, pos, bit)`

Sets or unsets a bit in an integer `x` at the given bit position `pos`.

Internally, `x` is first converted into its binary representation. Then `bit` is set to the `pos`-th position from the right of this binary representation of `x`. `bit` may be either **true** or **false**, or the numbers 0 or 1. E.g. if `x` is 2 = 0b0010, `pos` is 1, and `bit` is **true**, then the result is 3 = 0b0011.

`pos` should be an integer in the range $|pos| \in [1 .. 32]$.

See also: **getbit**, **getbits**, **setbits**, **setnbits**, **numarray.setbit**.

`setbits (x, r)`

Sets or unsets all 32 bits of an integer `x` with the bits given in register `r`. The register must contain a minimum of one, and a maximum of 32 values, either the Booleans **true** or **false**, or the integers 0 and 1. If the register contains less than 32 elements, and has length n, the first 32 - n bits `to the left` are *not* set. Example:

```
> setbits(8, reg(1, 0, 0)):
12
```

See also: **getbit**, **getbits**, **setbit**, **setnbits**, **numarray.setbit**.

`setmetatable (obj, metatable)`

Sets the metatable for the given table, set, sequence, or pair `obj`. (You cannot change the metatable of other types from Agena, only from C.) If `metatable` is **null**, removes the metatable of the given table. If the original metatable has a '__metatable' field, raises an error.

This function returns `obj`.

See also: **getmetatable**.

`setnbits (x, y [, pos [, nbits [, 'or']]])`

Sets `nbits` bits in 32-bit integer y into position `pos` of 32-bit integer `x`, and returns the modified value of `x`. `pos` and `nbits` should be in [1, 32]. If `pos` is not given, it is 1 by default (the right-most bit in `x`).

If `nbits` is not given, it is **math.mostsigbit**(`y`) by default.

By default, the bits in `x` are overwritten by the bits in `y`. If the fifth argument `'or'` (the string) is given, the bits are Boolean-OR'ed.

See also: **getbit**, **getbits**, **getnbits**, **setbit**, **setbits**.

```
settype (obj [, ···], str)
settype (obj [, ···], null)
```

In the first form the function sets the type of one or more procedures, sequences, tables, sets, pairs, or userdata `obj` to the name denoted by string `str`. **gettype** and **typeof** will then return this string when called with `obj`.

In the second form, by passing the **null** constant, the user-defined type is deleted, and **gettype** thus will return **null** whereas **typeof** will return the basic type of `obj`.

If `obj` has no `__tostring` metamethod, then Agena's pretty printer outputs the object in the form `str & '(' & <elements> & ')'` instead of the standard `'seq(' & <elements> & ')'` or `'<element>:<element>'` string.

If given just two arguments, i.e. an object and a string or an object and **null**, the function returns the modified object. In all other cases, the function returns **null**.

See also: **gettype**.

```
shift (obj, a, b)
```

Moves an element in table, sequence or register `obj` from position `old` to `new`, with `old`, `new` integers, shifting all the other elements accordingly - which might also cause a rotation. The function returns nothing.

See also: **move**, **purge**, **swap**.

```
size (obj)
```

With tables, the operator returns the number of key~value pairs in table `obj`.

With sets, pairs, and sequences, the operator returns the number of items in `obj`.

With registers, the operator returns the number of elements up to the current top pointer, but not the total number of elements in the registers.

With strings, the operator returns the number of characters in string `obj`, i.e. the length of `obj`.

See also: **environ.attrib**, **strings.strlen**, **strings.utf8size**, **tables.getsize**.

```
sort (obj [l [, u]] [, f])
```

Sorts table, sequence or register elements in a given order, in-place, from `obj[l]` to `obj[u]`, where by default `l` is 1 and `u` is the length of the structure. If `f` is given, then it must be a function that receives two structure elements, and returns **true** when

the first is less than the second (so that not `f(obj[i+1], obj[i])` will be **true** after the sort). If `f` is not given, then the standard operator `<` (less than) is used instead.

The sort algorithm is not stable; that is, elements considered equal by the given order may have their relative positions changed by the sort. Also, the function cannot sort structures featuring values of different types (see **skycrane.sorted** for an alternative). The return is the sorted structure.

See also: **sorted**, **stats.issorted**, **skycrane.sorted**, **stats.sorted**.

Example:

```
> s := [1, 2, 3]

> sort(s, << x, y -> x > y >>):
[3, 2, 1]

> s := seq(1:'a', 1.1:'b', 1.2:'c');

> sort(s, << x, y -> left(x) > left(y) >>):
seq(1.2:c, 1.1:b, 1:a)
```

sorted (obj [l [, u]] [, f])

Sorts table, sequence or register elements in `obj` in a given order from `obj[l]` to `obj[u]`, but - unlike sort - not in-place, and non-destructively. By default, `l` is 1 and `u` is the length of the structure. Depending on the type of `obj`, the return is a new table or sequence.

If `f` is given, then it must be a function that receives two structure elements to determine the sorting order. See **sort** for further information.

The function cannot sort structures featuring values of different types (see **skycrane.sorted** for an alternative).

See also: **sort**, **skycrane.sorted**, **stats.issorted**, **stats.sorted**.

subs (x:v [, ···], obj [, true])

Substitutes all occurrences of the value `x` in the table, set, sequence or register `obj` with the value `v`. More than one substitution pair can be given. The substitutions are performed sequentially and by default simultaneously starting with the first pair. The type of return is determined by the type of `obj`.

```
> subs(1:3, 2:4, [1, 2, -1]):
[3, 4, -1]
```

If present, the function also copies the metatable and user-defined type of `obj` to the new structure.

If the last argument is the option inplace=**true**, or the Boolean **true**, then the operation will be done in-place, modifying the original structure, but saving memory. After completion, the function returns the modified structure.

By default, **subs** is still replacing the elements in a structure with _all_ the replacements given, including intermediate substitutions. So if we have an expression like

```
> subs(1:2, 2:3, 3:4, [1, 2, 3])
```

we will get:

```
[4, 4, 4]
```

By passing the multipass=**false** option, the rest of the replacement list will be skipped as soon as a substitution has been done:

```
> subs(1:2, 2:3, 3:4, [1, 2, 3], multipass=false):
[2, 3, 4]
```

You can check numbers for approximate instead of strict equality by passing the new strict=**false** option.

See also: **countitems**, **map**, **remove**, **select**, **zip**.

sumup (obj)

Sums up all numeric values in table, sequence or register `obj`. The return is a number. If `obj` is empty or consists entirely of non-numbers, the operator returns **fail**. If the structure contains numbers and other objects, only the numbers are added. Numeric entries with non-numeric keys are processed, as well. The operator uses Kahan-Babuška Summation. To improve accuracy, you may sort `obj` before.

See also: **mulup**, **qsumup**, **calc.fsum**, **stats.cumsum**, **stats.sumdata**, **sort**, **sorted**.

swap (obj, a, b)

Swaps the table array, sequence or register `obj` entries at index positions `a` and `b`, with `a`, `b` integers. The function returns nothing.

See also: **move**, **purge**, **shift**.

time ()

Returns UTC time in seconds elapsed since the epoch in seconds as a number. The fractional part of the return represents milliseconds. The epoch usually is January 01, 1970, but this may vary between platforms.

See also: **os.clock**, **os.difftime**, **os.time**, **watch**, **skycrane.stopwatch**.

```
times (f, x, n [, ···])
```

```
times (f, x, infinity, eps [, ···])
```

In the first form, the function takes a start value `x` of any type, applies function `f` to it and repeatedly applies `f` to its previous result `n`-1 times. `n` should be a positive integer. It returns the result of the last call to `f`. The second and further arguments of `f` must be put right after `n`.

If `n` is less than 1, the function returns **null**.

Example:

```
> times(<< x -> 1 + recip x >>, 1, 33) -> 1.6180339887499  # Golden ratio
```

```
> f := << x -> times(<< n -> 0.5*(n + x/n)  >>, 1, 20) >> # square root
```

You can bail out of the loop prematurely by including a Boolean condition in the function definition. As soon as the expression evaluates to **false**, the iteration will stop and the previous interim result will be returned, e.g.:

```
> times( << x -> x < 10 and x + 1 >>, 1, 33):
10
```

If **times** should bail out in the first iteration then **false**, i.e. the result of the function call, will be returned:

```
> times( << x -> x < 0 and x + 1 >>, 1, 33):
false
```

In the second form, takes a start value `x` of any type, applies function `f` to it and repeatedly applies `f` to its previous result until the absolute difference of the last two function calls reaches or drops below the numeric threshold `eps`, a non-negative value.

If a function call evaluates to ±**infinity** or **undefined**, the operator also quits, returning **infinity** or **undefined**, respectively.

The third argument **infinity** just signals that the user wants to use this mode. If `f` is multivariate, all arguments but the first are passed right after `eps`.

Example: Solve the equation $7^3 + 2x - 5 = 0$ using Newton's method.

```
> import calc
```

```
> s := << x -> 7*x^3 + 2*x - 5 >>
```

```
> times(<< x -> x - s(x)/calc.eulerdiff(s, x) >>, 4, infinity, DoubleEps):
0.78792505251729
```

```
> s(ans):
0
```

See also: @ operator, **map**, **calc.aitken**.

top (obj)

With the table array, sequence or register `obj`, the operator returns the element with the largest index. If `obj` is empty, it returns **null**.

See also: **bottom**.

toreg (obj)

If `obj` is a string, the function will split it into its characters and return them in a register with each character in `obj` as a register value, and in the same order as the characters in `obj`.

If `obj` is a table, the function puts all its values - but not its keys - into a register.

If `obj` is a set, the function puts all its items into a register. The same applies to sequences.

If `obj` contains structures, then only their references are copied. Map **copy** to structures if you want to create independent copies of them.

In all other cases, the function issues an error.

See also: **toseq**, **toset**, **totable.**

toseq (obj)

If `obj` is a string, the function will split it into its characters and return them in a sequence with each character in `obj` as a sequence value, and in the same order as the characters in `obj`.

If `obj` is a table, the function puts all its values - but not its keys - into a sequence.

If `obj` is a set, the function puts all its items into a sequence. The same applies to registers.

If `obj` contains structures, then only their references are copied. Map **copy** to structures if you want to create independent copies of them.

In all other cases, the function issues an error.

See also: **toreg**, **toset**, **totable.**

`toset (obj)`

If `obj` is a string, the function will split it into its characters and returns them in a set. Note that there is no order in the resulting set.

If `obj` is a table, sequence or register, the function puts all its values - but not its keys - into a new set.

If `obj` contains structures, then only their references are copied. Map **copy** to structures if you want to create independent copies of them.

In all other cases, the function issues an error.

See also: **toreg**, **toseq**, **totable**.

`totable (obj)`

If `obj` is a string, the function splits it into its characters, and returns them in a table with each character in `obj` as a table value in the same order as the characters in `obj`.

If `obj` is a sequence, register, or set, the function converts it into a table.

If `obj` contains structures, then only their references are copied. Map **copy** to structures if you want to create independent copies of them.

In all other cases, the function issues an error.

See also: **toreg**, **toseq**, **toset**.

`type (obj)`

This operator returns the basic type of its only argument `obj`, coded as a string. The possible results of this function are `'null'` (the string, not the value **null**), `'number'`, `'string'`, `'boolean'`, `'table'`, `'set'`, `'sequence'`, `'register'`, `'pair'`, `'complex'`, `'procedure'`, `'thread'`, `'lightuserdata'`, and `'userdata'`.

If `obj` is a table, set, sequence, pair, or procedure with a user-defined type, then **type** always returns the basic type, e.g. `'sequence'` or `'procedure'`.

See also: **::** and **:-** operators, **checktype**, **gettype**, **typeof**.

`typeof (obj)`

This operator returns the user-defined type - if it exists - of its only argument `obj`, coded as a string.

A self-declared type can be defined for procedures, tables, pairs, sets, and sequences with the **settype** function. If there is no user-defined type for `obj`, then the basic type will be returned, i.e. `'null'` (the string, not the value **null**), `'number'`, `'string'`, `'boolean'`, `'table'`, `'set'`, `'register'`, `'sequence'`, `'pair'`, `'complex'`, `'procedure'`, `'thread'`, and `'userdata'`.

See also: **::** and **:-** operators, **type**, **gettype**.

unassigned (obj)

This Boolean operator checks whether an expression `obj` evaluates to **null**. If `obj` is a constant, i.e. a number, boolean including **fail**, or a string, the operator always returns **false**.

See also: **assigned**.

unique (obj)

With a table `obj`, the function removes all holes (`missing keys`) and removes multiple occurrences of the same value in the array part, if present. The hash part of a table is considered to be always unique by definition, so it just copies it to the result. The return is a new table with the original table unchanged.

With a sequence or register `obj`, the **unique** function removes multiple occurrences of the same value, if present. The return is a new sequence or register with the original structure unchanged.

See also: **tables.entries**.

unpack (obj, [, i [, j]])

Returns the elements from the given table, sequence or register `obj`. This function is equivalent to

```
    return obj[i], obj[i+1], ···, obj[j]
```

except that the above code can be written only for a fixed number of elements. By default, `i` is 1 and `j` is the length of the object, as defined by the **size** operator.

Please note that if you put a call to **unpack** into an expression list, only the first return of **unpack** is propagated if the call to **unpack** is not at the final position of the expression list, for example:

```
> s := [unpack([1, 2, 3]), 4, 5]:   # 2 and 3 are discarded
[1, 4, 5]

> s := [-1, 0, unpack([1, 2, 3])]:  # 2 and 3 are included
[-1, 0, 1, 2, 3]
```

Consider **copyadd** in this situation. See also: **identity**, **ops**, **values**.

`values (obj, i₁ [, i₂, ···]])`

Returns the elements i_k from the given table, sequence or register `obj`. This function is equivalent to - for example -

```
return [ i₁ ~ obj[i₁], i₂ ~ obj[i₂], ··· ]  or
return seq( obj[i₁], obj[i₂], ··· )
```

The type of return is determined by the first argument `obj`.

See also: **columns**, **ops**, **select**, **unpack**.

`watch ([option])`

The function implements a stop watch. With the first call, the function starts counting and returns 0. The second call returns the elapsed time in seconds and milliseconds and restarts the clock. If any argument is given, then the clock is reset, but it does not start counting.

See also: **time**, **os.time**, **skycrane.stopwatch**.

`whereis (obj, x)`

Returns the indices for a given value `x` in table, sequence or register `obj` as a new table, sequence or register, respectively, dependent on the type of `obj`.

See also: **tables.entries**, **tables.indices**.

`write ([fh,] v₁ [, v₂, ···] [, delim = <str>])`

This function prints one or more numbers, Booleans or strings `vₖ` to the file denoted by the handle `fh`, or to stdout (i.e. the console) if `fh` is not given.

By default, no character is inserted between neighbouring values. This may be changed by passing the option `'delim':<str>` (e.g. `'delim':'|'` or `delim='|'`) as the last argument to the function with `<str>` being a string of any length. Remember that in the function call, a shortcut to `'delim':<str>` is `delim = <str>`.

The function is an interface to **io.write**.

See also: **printf**, **skycrane.scribe**, **skycrane.tee**.

`writeline ([fh,] v₁ [, v₂, ···] [, delim = <str>])`

This function prints one or more numbers, booleans or strings `vₖ` followed by a newline to the file denoted by the handle `fh`, or to stdout (i.e. the console) if `fh` is not given.

By default, no character is inserted between neighbouring values. This may be changed by passing the option `'delim':<str>` (i.e. a pair, e.g. `'delim':'|'`) as the last argument to the function with `<str>` being a string of any length. Remember that in the function call, a shortcut to `'delim':<str>` is `delim = <str>`.

The function is an interface to **io.writeline**.

See also: **printf**, **skycrane.scribe**, **skycrane.tee**.

`xpcall (f, err [, arg1, ···])`

This function is similar to **protect**, except that you can set a new error handler.

xpcall calls function `f` in protected mode, using `err` as the error handler. Arguments to `f` are optional. Any error inside `f` is not propagated; instead, **xpcall** catches the error, calls the `err` function with the original error object, and returns a status code. Its first result is the status code (a Boolean), which is **true** if the call succeeds without errors. In this case, **xpcall** also returns all results from the call, after this first result. In case of any error, **xpcall** returns **false** plus the result from `err`.

See also: **protect**.

`zip (f, obj1, obj2 [, ···])`

`zip (op, obj1, obj2)`

In the first form, the function zips together either two sequences, two registers, or two tables `obj1`, `obj2` by applying the function `f` to each of its respective elements. Depending on the type of `obj1`, `obj2`, the result is a new sequence, register, or table `s` where each element `s[k]` is determined by `s[k] := f(obj1[k], obj2[k])`.

`obj1` and `obj2` must have the same number of elements. If you pass tables, they must have the same keys.

If `f` has more than two arguments, then its third to last argument must be given right after `B`.

In the second form, `op` depicts an arithmetic operator, represented as a string, that zips together the structure elements:

- `'+'`: addition,
- `'-'`: subctraction,
- `'*'`: multiplication,
- `'/'`: division,
- `'\\'`: integer division,
- `'%'`: modulus,
- `'^'`: exponentiation,
- `'**'`: integer exponentiation.

This method is twice as fast with sequences and registers, and 50 % faster with tables.

If `obj1` or `obj2` have user-defined types or metatables, they are copied to the resulting structure, as well. If `obj1` has a metatable, then this metatable is copied, else the metatable of `obj2` is used if the latter exists. The same applies to user-defined types.

See also: **map**, **remove**, **select**, **subs**.

Chapter **Nine**

Strings

9 Strings

9.1 Basic String Functions

Summary of Functions:

Search

> **atendof, has, in, notin, instr, strings.find, strings.glob, strings.match, strings.mfind**.

Insertion, Substitution, and Deletion

> **replace, strings.appendmissing, strings.chomp, strings.chop, strings.gsub, strings.include, strings.remove, strings.wrap, strings.wrapmissing**.

Extraction

> **split, strings.advance, strings.between, strings.charset, strings.fields, strings.gmatch, strings.gmatches, strings.gseparate, strings.separate, strings.splitfields, strings.strchr, strings.strrchr, strings.strstr**.

Queries

> **abs, empty, filled, strings.charmap, strings.compare, strings.contains, strings.diffs, strings.dleven, strings.fuzzy, strings.isaligned, strings.isalpha, strings.isalphanumeric, strings.isalphaspace, string.isalphaspec, strings.isascii, strings.isblank, strings.iscenumeric, strings.iscontrol, strings.isdia, strings.isending, strings.isfloat, strings.isgraph, strings.ishex, strings.islatin, strings.isisoalpha, strings.isisolower, strings.isisoprint, strings.isisospace, strings.isisoupper, strings.islatinnumeric, strings.isloweralpha, strings.islowerlatin, strings.ismagic, strings.isnumber, strings.isnumeric, strings.isnumberspace, strings.isprintable, strings.isspace, strings.isspec, strings.isstarting, strings.isupperalpha, strings.isupperlatin,strings.isutf8, strings.iswrapped, strings.shannon, strings.strcmp, strings.strverscmp**.

Counting

> **size, strings.hits, strings.strlen, strings.utf8size, strings.words**.

Formatting

> **lower, trim, upper, strings.align, strings.capitalise, strings.format, strings.isolower, strings.isoupper, strings.ljustify, strings.ltrim, strings.lrtrim, strings.rjustify, strings.rtrim, strings.tolower, strings.toupper, strings.uncapitalise**.

Conversion

> **&, join, tonumber, tostring, strings.a64, strings.diamap, strings.iterate, strings.pack, strings.packsize, strings.reverse, strings.tolatin, strings.toutf8, strings.transform, strings.unpack**.

Manipulation

> **@, map, strings.iterate, strings.repeat, strings.rotateleft, strings.rotateright, strings.tobytes, strings.tochars**.

Miscellaneous

> **strings.random**.

A note in advance: All operators and **strings** package functions know how to handle many diacritics properly. Thus, the **lower** and **upper** operators know how to convert these diacritics, and various **is*** functions recognise diacritics as alphabetic characters.

Diacritics in this context are the letters:

```
â  Â  ä  Ä  à  À  á  Á  å  Å  æ  Æ  ã  Ã
ê  Ê  ë  è  È  é  É  Ë
ï  Ï  î  Î  ì  Ì  í  Í  ý  Ý  ÿ
ô  Ô  ö  Ö  ò  Ò  ø  Ø  ó  Ó  õ  Õ
û  Û  ù  Ù  ü  Ü  ú  Ú
ç  Ç  ñ  Ñ  ð  Ð  þ  Þ  ß
```

9.1.1 Operators and Functions

`s1 & s2`

This binary operator concatenates two strings `s1`, `s2` and returns a new string. `s1` or `s2` may also be a number or a Boolean; in this case the argument will be converted to a string and then concatenated with the other operand.

See also: **join**.

`v &:= s`

The compound concatenation operator appends string `s` to the contents of the string variable `v`. It is equivalent to: `v := v` **&** `s`.

`s1 atendof s2`

This binary operator checks whether string `s2` ends in a substring `s1`. If true, the position of the position of `s1` in `s2` will be returned; otherwise **null** will be returned. The operator also returns **null** if the strings have the same length or at least one of them is the empty string.

See also: **in**, **instr**, **strings.isstarting**, **strings.isending**.

`s1 in s2`

This binary operator checks whether string `s2` includes `s1` and returns its position as a number, or **null** if `s1` cannot be found. The operator also returns **null** if at least one of the strings is the empty string.

See also: **atendof**, **instr**, **notin**, **strings.contains**, **strings.isstarting**, **strings.isending**.

`s1 notin s2`

This binary operator checks whether string `s2` does not include `s1` and returns **true** or **false**.

See also: **in** operator.

`s1 split s2`

Splits the string `s1` into words. The delimiter is given by string `s2`, which may consist of one or more characters. The return of the operator is a sequence. If `s1`= `s2`, or if `s2` is the empty string, then an empty sequence will be returned.

See also: **strings.fields**, **strings.iterate**, **strings.separate**.

`abs (s)`

With strings, the operator returns the numeric ASCII value of the given character `s` (a string of length 1).

`empty (s)`

The operator checks whether the string `s` is empty. The return is **true** or **false**. See also: **filled**.

`filled (s)`

The operator checks whether the string `s` is non-empty. The return is **true** or **false**. See also: **empty**.

`has (s, chars)`

Checks whether at least one character in `s` matches one of the characters in `chars`, a string representing a set of individual characters.

See also: **in** operator, **strings.contains**.

`instr (s, pattern [, init] [, plain] [, 'reverse'] [, 'borders'])`

Looks for the first match of string `pattern` in the string `s`. If it finds a match, then **instr** returns the index of `s` where this occurrence starts; otherwise, it returns **null**.

If `pattern` is a set of pattern strings, returns **true** if at least one of the patterns matches `s`; otherwise returns **false**.

If the option `'reverse'` is given, then the search starts from the right end and always runs to its left beginning and the first occurrence of `pattern` with respect to the beginning of `s` will be returned. In the reverse search, pattern matching is not supported.

An optional numerical argument `init` passed anywhere after the second argument specifies where to start the search; its default value is 1 and may be negative. In the latter case, the search is started from the `|init|`'s position from the right end of `s`.

The function by default supports pattern matching, almost similar to regular expressions, see Chapter 9.1.3. **instr** is 45 % faster than **strings.find**. If the optional Boolean argument `plain` is set to the Boolean **true**, pattern matching is switched off and a much faster plain search is conducted instead (speed bonus around 40 %).

The optional argument `'borders'` returns the start and the end position of a match in a pair. However, this mode is slow, use **strings.find** instead which is twice as fast.

See also: **atendof**, **in**, **strings.isstarting**, **strings.isending**, **strings.find**.

`join (obj [, sep [, i [, j]]])`

Concatenates all string values in the table, sequence or register `obj` in sequential order and returns a string: `obj[i]` & `sep` & `obj[i+1]` ⋯ & `sep` & `obj[j]`. The default value for `sep` is the empty string, the default for `i` is 1, and the default for `j` is the length of the sequence. The function issues an error if `obj` contains non-strings.

See also: **&** operator.

lower (s)

The operator receives a string and returns a copy of this string with all uppercase letters ('A' to 'Z' plus the above mentioned diacritics) changed to lowercase ('a' to 'z' and the diacritics listed at the end of Chapter 9.1). The operator leaves all other characters unchanged. Example:

```
> lower('Elektronika MK-61'):
elektronika mk-61
```

See also: **strings.isolower**, **strings.tolower**, **upper**.

map (f, s [, ···] [, true])

This function maps a function `f` to all characters of string `s` from the left to right. The return is a sequence of function values.

If function `f` has only one argument, then only the function and the string `s` must be passed to **map**. If the function has more than one argument, then all arguments *except the first* are passed right after argument `s`. If the last argument is the option inplace=**true**, or the Boolean **true**, then the operation will be done in-place, modifying the original structure, but saving memory. After completion, the function returns the modified structure.

replace (s1, s2, s3)

replace (s1, obj)

replace (s1, pos, s2)

In the first form, the function replaces all occurrences of string `s2` in string `s1` by string `s3`.

In the second form, the function receives a string `s1` and a table, sequence or register `obj` of one or more string pairs of the form `s2`:`s3` and replaces all occurrences of `s2` in string `s1` with the corresponding string `s3`. Thus you can replace multiple patterns simultaneously with only one call to **replace**.

In the third form, the function inserts a new string `s2` into the string `s1` at the given position `pos`, substituting the respective character in `s1` with the new string `s2` which may consist of zero, one or more characters. The return is a new string. If `s2` is the empty string, the character in `s1` is deleted. The return is always a new string.

The function does not support pattern matching, use **strings.gsub** instead.

See also: **utils.singlesubs**.

`size (s)`

With a string `s`, the operator returns its length, i.e. the number of characters in `s`.

`tonumber (e [, base])`

Tries to convert its argument to a number or complex value. If the argument is already a number, complex value, or a string convertible to a number or complex value, then **tonumber** returns this value; otherwise, it returns `e` if `e` is a string, and **fail** otherwise. The function recognises the strings `'undefined'` and `'infinity'` properly, i.e. it converts them to the corresponding numeric values **undefined** and **infinity**, respectively.

An optional argument specifies the base to interpret the numeral. The base may be any integer between 2 and 36, inclusive. In bases above 10, the letter 'A' (in either upper or lower case) represents 10, 'B' represents 11, and so forth, with 'Z' representing 35. In base 10 (the default), the number may have a decimal part, as well as an optional exponent part. In other bases, only unsigned integers are accepted. If an option is passed, `'undefined'` and `'infinity'` are not converted to numbers; and if `e` could not be converted, **fail** will be returned.

`tostring (e [, anyoption])`

Receives an argument `e` of any type and converts it to a string in a reasonable format. For complete control of how numbers are converted, use **strings.format**.

If the metatable of `e` has a '__tostring' field, then the **tostring** function calls the corresponding value with e as argument, and uses the result of the call as its result. With numbers, the number of digits in the resulting string is dependent on the **kernel/digits** setting. See **environ.kernel** for further information.

If `e` is a complex number, its real and imaginary parts are returned as two strings. If any option is given, the return is one string of the format ±"re+im*I" or ±"re-im*I", depending on the sign of the imaginary part of `e`.

See also: **tostringx**.

`tostringx (e)`

Works like **tostring** but also formats structures, userdata and complex numbers the same way as the prettyprinter does, or in other words: it returns the argument as a string formatted the same way as the **print** function outputs it on screen. This is useful if you want to write structures or complex numbers to a file.

`trim (s)`

Returns a new string with all leading, trailing and excess embedded white spaces removed. **trim** is an operator. See also: **strings.ltrim**, **strings.lrtrim**, **strings.rtrim**.

upper (s)

The operator receives a string and returns a copy of this string with all lowercase letters ('a' to 'z' plus the above mentioned diacritics) changed to uppercase ('A' to 'Z' and the diacritics listed at the end of Chapter 9.1). The operator leaves all other characters unchanged. Example:

```
> upper('Elektronika MK-61'):
ELEKTRONIKA MK-61
```

See also: **lower**, **strings.capitalise**, **strings.isoupper**, **strings.toupper**.

9.1.2 The strings Library

The **strings** library provides generic functions for string manipulation, such as finding and extracting substrings, and pattern matching. When indexing a string in Agena, the first character is at position 1 (not at 0, as in C). Indices are allowed to be negative and are interpreted as indexing backwards, from the end of the string. Thus, the last character is at position -1, and so on.

The strings library provides all its functions inside the table `strings`.

strings.a64 (x)

The function converts between 32-bit long integers and little-endian base-64 ASCII strings (of length 0 to 6).

If the argument `x` is a base-64 ASCII string, the result is a signed 32-bit integer; if the argument `x` is a number, the result is the base-64 ASCII string, which consists of the characters:

./0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz

strings.advance (s, p [, option])

strings.advance (s, pos)

In the first form, the function moves to substring `p` in string `s` and returns `p` up to the end of `s`. If `p` could not be found, the function returns **null**. The function supports pattern matching. If the optional third argument **true** is given, the function returns the rest of `s` following but not including `p`. In this case, if `s` ends with `p`, **null** will be returned.

In the second form, the substring starting at position `pos` (a positive integer) up to the end of `s` will be returned. If `pos` is greater than the length of `s`, the result is **null**.

See also: **strings.find**.

`strings.align (s [, n])`

Inserts newlines into a string `s` after each `n` character. By default `n` is 79, so a newline is inserted at position 80, 160, and so forth. The return is a string. The function helps with correctly outputting formatted text at the console.

`strings.appendmissing (s, t)`

Appends suffix `t` (a string) to `s` (a string) if `t` is not already at its end; otherwise returns `s`. If `s` is the empty string, `t` will be returned.

`strings.between (s, p, q [, true])`

Returns the substring in string `s` that is nested between the prefix string `p` and the suffix string `q`. `p` or `q` may reside within the string. If the Boolean value **true** is given as a fourth argument, the function tries to return the substring found as a number. If nothing could be found, the function returns **null**.

See also: **strings.chomp**, **strings.include**.

`strings.capitalise (s)`

Converts the first character in string `s` to upper case - if possible - and returns the capitalised string. If `s` is the empty string, it is simply returned. It also converts ligatures if the Western European character set is being used.

See also: **upper**, **strings.uncapitalise**.

`strings.charmap ()`

Queries the internal tables to classify characters. Returns a table of key~value pairs:

Key	Sequence of	Used by/Comment
alpha	alphabetical letters	strings.alpha, strings.isalphanumeric, strings. isalphaspace, strings.isalphaspec
ascii	table of all ASCII characters	result may vary across platforms and codepages
upper	upper-case letters	strings.isupperalpha
lower	lower-case letters	strings.isloweralpha
vowel	vowels	strings.isvowel
dia	diacritics	strings.isalpha, strings.isalphanumeric, strings.isalphaspace, strings.isalphaspec, strings.isdia
digits	digits 0 to 9	strings.isalphanumeric
hex	hexadecimals	strings.ishex
punct	punctuations	strings.isspec, strings.isalphaspec

Key	Sequence of	Used by/Comment
`control`	control characters such as \n, \r, \b	strings.iscontrol
`blank`	white space, tab	strings.isblank
`printable`	printable characters	strings.isprintable

`strings.charset (s)`

Returns a set of all the unique characters included in a string.

See also: **has**, **strings.contains**.

`strings.chomp (s, t [,···])`

Removes pattern string `t` from the end of string `s` if it is there, and returns the shortened string `s`; otherwise returns `s` unchanged. If more than one pattern is given, each additional pattern will be removed from the previous result.

The function supports pattern matching. In this case you or may not terminate the pattern with one final '`$`'.

Example:

```
> strings.chomp('agena language', '(%a+)', 'ena '):
ag
```

See also: **strings.chop**.

`strings.chop (s)`
`strings.chop (s, f)`

In the first form, removes the last character from string `s` and returns the shortened string. If `s` is empty, it is simply returned.

In the second form, if a function `f` returning **true** or **false** is given, **chop** checks each character in the string from the right to the left for the given Boolean condition and returns the string from its beginning up to and including the character that no longer satisfies the condition.

Example:

```
> strings.chop('path/file.name', << x -> x <> '/' >>):
path/
```

See also: **strings.between**, **strings.chomp**.

`strings.compare (s1, s2)`

When called with no option, returns the first position - an integer - where the two strings `s1` and `s2` differ, or 0 if both strings are equal.

See also: **strings.strcmp**, **strings.strverscmp**.

`strings.contains (s, t)`

Checks whether all characters in string `s` are part of the characters in string `t`, and returns **true** or **false**.

See also: **has**, **in** operator, **strings.charset**.

`strings.cut (s, d)`

The function takes a string `s` to be split into two pieces, and a string `d` of one or more single-character delimiters, and returns two values: the first part of `s` up to - but not including - the delimiter found, and the rest of `s` also without the delimiter.

If a string cannot be split apart, it will be returned as the first result and the second return is **null**.

See also: **split**, **strings.separate**.

`strings.diffs (s, t [, n [, option]])`

Counts the differences between the two strings `s` and `t`: substitutions, transpositions, deletions, and insertions.

By default, both strings must contains at least `n`=3 characters. You may change this by passing any other positive number for `n`. The function returns **fail** if at least one of the strings consists of less characters.

If any fourth argument is given, the return is a sequence of strings describing the respective difference found, otherwise the returns is the number of the differences encountered.

The function is at least thrice as fast as **strings.dleven**, but may count differently in odd situations.

See also: **strings.dleven**, **strings.fuzzy**, **skycrane.tolerance**.

`strings.dleven (s, t)`

Returns the Damerau-Levenshtein distance between two strings `s` and `t`. It is a count of the minimum number of insertions, deletions, substitutions of a single character, or transpositions of two neighbouring characters to convert `s` into `t`. The return is a number. If at least one of the strings is empty, **undefined** will be returned.

See also: **strings.fuzzy**, **strings.diffs**, **skycrane.tolerance**.

`strings.diamap (s [, option])`

The function corrects problems in the Solaris, Linux, OS/2, Windows, and DOS consoles running code page 850 with diacritics and ligatures read in from the keyboard or a text file by mapping them to code page 1252. It takes a strings `s`, applies the mapping, and returns a new string. All other characters are returned unchanged.

If any `option` is given, the function transforms a string from code page 1252 to 850. Example:

```
> strings.diamap('AEIOU-Í_ã+Ï'):
AEIOUÄÖÜÆÅØ
```

Note that the function does not convert all existing special tokens.

Agena is shipped with substitution tables for code page 1252. If you want to use another code page, edit the `_c2f` and `_f2c` tables in the lib/library.agn file accordingly.

See also: **os.codepage**.

`strings.dump (f [, strip])`

Returns a string containing a binary representation of the given function `f`, so that a later **loadstring** on this string returns a copy of the function. `f` must be an Agena function without upvalues, remember table or internal store table.

If `strip` is a **true** value, the binary representation may not include all debug information about the function, to save space.

The function can also be used to binarily serialise data by defining a function returning data, e.g.:

```
> f := proc() is return [1, 2, 3] end;

> s := strings.dump(f);

> loadstring(s)():
[1, 2, 3]
```

See also: **strings.tobytes**.

```
strings.fields (s, i₁ [, i₂, ···] [, delim] [, true])
strings.fields (s, o [, delim] [, true])
```

Extracts the given fields (columns) in string `s`. In the first form, the field positions i_1, i_2, etc. are non-zero integers. The field positions may be negative, denoting fields counted from the right end of `s`. In the second form, the field positions are given in the sequence `o`.

An optional string `delim` may be passed to denote the character or character sequence that separates the individual fields. The default for `delim` is the white space. If the Boolean value **true** is given as the last argument, the function tries to convert the fields into numbers.

The return is a sequence of the fields (strings).

See also: **strings.iterate**, **strings.gseparate**, **split**, especially if you want to retrieve all fields in a string.

```
strings.find (s, pattern [, init [, plain]])
```

Looks for the first match of string `pattern` in string `s`. If it finds a match, then **find** returns the indices of `s` where this occurrence starts and ends; otherwise, it returns **null**. The function does support pattern matching facilities (which you can turn off, see below). If `pattern` is a table, set, sequence or register of string patterns, then the function checks whether at least one of the patterns matches `s` and returns the respective result.

A third, optional numerical argument `init` specifies where to start the search; its default value is 1 and may be negative. A value of **true** as a fourth, optional argument `plain` turns off the pattern matching facilities (see Chapter 9.1.3), so the function does a plain `find substring` operation, with no characters in pattern being considered `magic`. Note that if `plain` is given, then `init` must be given as well.

If the pattern has captures, then in a successful match the captured values are also returned, after the two indices.

See also: **in**, **atendof**, and **instr**, **strings.mfind**.

```
strings.format (formatstring, ···)
```

Returns a formatted version of its variable number of arguments following the description given in its first argument (which must be a string). The format string almost follows the same rules as the ISO C function `sprintf`. The only differences are that the conversion specifiers *, I and L are not supported and that there are thirteen extra specifiers: a, A, b, B, h, H, m, n, N, p, P, q, Q, D, and F.

For an overview of all available specifies and examples, see below.

In general a format has the following syntax, where values in square brackets are optional:

%[flags][width][.precision]

'flags' may be one of the specifiers described below, optionally preceded by:

- – (minus):   left-justify the result,
- + (plus):   print plus sign in front of positive numbers

You can mix – and +.

'width' is the minimum length of the output in characters. 'precision' depicts the minimum number of decimal places to appear, with trailing zeros to be added if necessary.

The following specifiers do not comply to the C standard:

The q specifier formats a string in a form suitable to be safely read back by the Agena interpreter: All double quotes, newlines, embedded zeros, and backslashes in the string are correctly escaped when written, and without trailing zeros in the fractional part when the precision specified in the specifier is greater than the number of significant digits in the argument supplied. The same applies to Q but with single quotes. The a and A specifiers work the same like the q and Q specifiers, respectively, but do not include trailing or leading double quotes. The B specifier prints a string in backquotes. The b specifier prints a Binary value.

For instance, the call

```
> strings.format('%q', 'a string with \"quotes\" and \n new line')
```

will produce the string:

```
 "a string with \"quotes\" and \
 new line"
```

The h and H specifiers print a floating-point number in a hexadecimal fractional notation with the exponent to base 2 represented in decimal digits. On DOS and OS/2, the h and H specifiers are not available, and in Windows 2000 they do not work.

The p specifier multiplies the given number by 100 and displays it in fixed float ('f') format, followed by a percent sign. The m specifier prints a monetary amount with thousands separators and the decimal point defined by the current locale, the default is the format string '%.2f'.

The specifier P formats the pointer (returned by lua_topointer). That gives a unique string identifier for structures, userdata, threads, strings, and functions. For other values (numbers, **null**, booleans), this specifier results in a string representing the pointer NULL.

The n and N specifiers print a number using the decimal point separator of the locale of the operating system (which may differ from the locale in use by Agena), otherwise they work like the f and F specifiers.

The specifiers D and F prevent quarrels with numerical functions that may return non-numbers in case of errors: D formats an integer like the d specifier if the argument is a number, and the C double representation of **undefined** otherwise if the value is not a number. Likewise, F and N either format a float, or the C double companion piece of **undefined** (e.g. `1.#QNAN0` in Windows) if the value is not a number.

The conversion specifiers c, D, d, E, e, f, F, g, G, h, H, i, m, n, N, o, p, u, X, and x all expect a number as argument, whereas s expects a string, and a, A, P, q, Q and expect anything.

This function does not accept string values containing embedded zeros.

Examples:

```
> strings.format('%+15.9f', 10k*Pi):
+31415.926535898

> strings.format('%15.9f', 3.5):
    3.500000000

> strings.format('%-15.9f', 3.5):
3.500000000

> strings.format('%015.9f', 3.5):
00003.500000000

> strings.format('%15.0f', 3.5):
              4

> strings.format('%15d', 3.5):
              3

> strings.format('%X', 2^16-1):
FFFF

> strings.format('%c', 97):
a

> strings.format('%s', 'agena >>'):
agena >>

> strings.format('%q', 'agena >>'):
"agena >>"
```

```
> strings.format('%d\n%2d\n%02d\n%2.5f\n%+2.5f\n+2.5f\n%s', 1, 1, 1,
>    Pi, Pi, -Pi, 'New Horizons'):
1
 1
01
3.14159
+3.14159
-3.14159
New Horizons
```

Summary:

Specifier	Description	Example
%d, %i	writes as an integer	-1, 1
%o	writes as an octal number	12
%b	writes a binary number in the range [-1023, 1023]	-0b1111111111
%u	writes as an unsigned integer, with a cast to C's uint32_t	10
%x, %X	writes as unsigned hexadecimal number, with a cast to uint32_t. %x uses lower-case, %X upper-case	f, F
%f	writes as a floating-point number in normal, fixed-point notation	3.141593
%lf	writes as a floating-point number in normal, fixed-point notation with 16 fractional digits by default	3.1415926535..
%ld	writes a long double in normal, fixed-point notation with 19 fractional digits by default, see **long** package in Chapter 11.15	
%e, %E	writes a floating-point number in exponential notation. %e uses lower-case, %E upper-case	3.141593e+000, 3.141593E+000
%g, %G	writes a floating-point number in either normal or exponential notation, depending on its magnitude. %g uses lower-case, %G upper-case	1e-006 1E-006
%h, %H	writes a floating-point number in a hexadecimal fractional notation with the exponent to base 2 represented in decimal digits. %h uses lower-case, %H upper-case	0x1.921fb5p+1, 0X1.921FB5P+1
%p	writes in percent	314.159265%
%m	writes a monetary amount with thousands separators and the decimal point defined by the current locale	31,415.93
%n, %N	writes a number using the decimal point separator of the locale of the operating system	31415,926536
%D	writes an integer if the value is a number, and "undefined" otherwise	3, undefined
%F	writes a floating-point number if the value is a number, and "undefined" otherwise	3.141593, undefined

Specifier	Description	Example
%q, %Q, %B	writes a string put in double (%q), single (%Q) or backquotes (%B) suitable to be safely read back by the interpreter	"3.1415926535.."
%a, %A	like %q but without surrounding quotes	3.1415926535..
%c	writes the corresponding ASCII character	a
%s	writes a string, numbers are automatically converted properly	agena, 3.1415926535..
%%	writes percentage sign	%

strings.fuzzy (s, t)

Compares two strings case-insensitively and returns an estimate of their similarity as both an absolute and relative score, the latter taking into account the length of the longer string.

One point is given for a matching character. Subsequent matches are given two extra points. A higher score indicates a higher similarity. With the second return, 1 depicts equality, and a lower value the degree of similarity.

The function is written in Agena and included in the lib/library.agn file.

See also: **strings.dleven**, **strings.diffs**, **skycrane.tolerance**.

strings.glob (s, pattern [, true])

Compares a string `s` with a string `pattern`, the latter optionally including the wildcards `?` and `*`, where `?` represents exactly one unknown character, and `*` represents zero or more unknown characters. Other pattern matching facilities are not supported.

The return is **true** if the pattern could be found, and **false** otherwise. If the optional third argument is **true**, then the strings are compared case-insensitively.

See also: **strings.find**.

strings.gmatch (s, pattern)

Returns an iterator function that, each time it is called, returns the next captures from `pattern` over string `s`. The function supports pattern matching facilities described in Chapter 9.1.3.

If `pattern` specifies no captures, then the whole match is produced in each call.

As an example, the following loop

```
> s := 'hello world from Lua'
```

```
> for w in strings.gmatch(s, '%a+') do
>    print(w)
> od
```

will iterate over all the words from string `s`, printing one per line. The next example collects all pairs key~value from the given string into a table:

```
> create table t;

> s := 'from=world, to=Lua'

> for k, v in strings.gmatch(s, '(%w+)=(%w+)') do
>    t[k] := v
> od
```

See also: **strings.match**, **strings.gmatches**.

strings.gmatches (s, pattern)

Wrapper around **strings.gmatch** which returns all occurrences of a substring `pattern` in string `s` a in a new sequence.

The function is written in Agena and included in the lib/library.agn file.

strings.gseparate (s, pattern [, tonumber [, init]])

The function takes a string `s` to be split apart into its tokens one after another, and a delimiter string `pattern`, and returns an iterator function that each time it is called, returns one token. If the end of `s` has been reached, the function returns **null**. The function supports pattern matching. For an iterator without pattern matching, see **strings.iterate**.

If `s` starts with the delimiter, an empty string will be returned.

If the Boolean value **true** is passed to `tonumber`, the function tries to return a number. If `init`, a positive integer is given, the function searches from the `init`'th character in `s`.

If called with any argument, the function returns the number of tokens returned but does not search for the next token.

See also: **split**, **strings.fields**, **strings.iterate**, **strings.separate**.

strings.gsub (s, pattern, repl [, n])

Returns a copy of `s` in which all occurrences of the `pattern` have been replaced by a replacement string specified by `repl`, which may be a string, a table, or a function. **gsub** also returns, as its second value, the total number of substitutions made. See Chapter 9.1.3 for more information on patterns.

If `repl` is a string, then its value is used for replacement. The character % works as an escape character: any sequence in `repl` of the form %n, with n between 1 and 9, stands for the value of the n-th captured substring (see below). The sequence %0 stands for the whole match. The sequence %% stands for a single %.

If `repl` is a table, then the table is queried for every match, using the first capture as the key; if the pattern specifies no captures, then the whole match is used as the key.

If `repl` is a function, then this function is called every time a match occurs, with all captured substrings passed as arguments, in order; if the pattern specifies no captures, then the whole match is passed as a sole argument.

If the value returned by the table query or by the function call is a string or a number, then it is used as the replacement string; otherwise, if it is **false** or **null**, then there is no replacement (that is, the original match is kept in the string).

The optional last parameter `n` limits the maximum number of substitutions to occur. For instance, when `n` is 1 only the first occurrence of pattern is replaced.

Here are some examples:

```
x := strings.gsub('hello world', '(%w+)', '%1 %1')
--> x = 'hello hello world world'

x := strings.gsub('hello world', '%w+', '%0 %0', 1)
--> x = 'hello hello world'

x := strings.gsub('hello world from Lua', '(%w+)%s*(%w+)', '%2 %1')
--> x = 'world hello Lua from'

x := strings.gsub('home = $HOME, user = $USER', '%$(%w+)', os.getenv)
--> x = 'home = /home/roberto, user = roberto'

x := strings.gsub('4+5 = $return 4+5$', '%$(.-)%$', proc (s)
return loadstring(s)()
end)
--> x = '4+5 = 9'

local t := [name~'lua', version~'5.1']
x = strings.gsub('$name%-$version.tar.gz', '%$(%w+)', t)
--> x = 'lua-5.1.tar.gz'
```

See also: **replace**.

strings.hits (s, pattern [, true])

Returns the number of occurrences of substring `pattern` in string `s`.

If only two arguments are passed, pattern matching facilities (see Chapter 9.1.3) are supported. If the Boolean constant **true** is passed as a third argument, pattern matching is switched off for faster execution.

See also: **strings.words**.

```
strings.include (s, pos, p)
```
Inserts the string `p` into the string `s` at position `pos`.

If `pos` ≤ **size** `s`, the character at position `pos` is moved size `p` places to the right.

If `pos` = **size** `s` + 1, `p` is just appended to `s`, equal to the Agena expression `s` **&** `p`.

The function returns the new string and issues an error, if the index `pos` is invalid. p may be the empty string, in this case, `p` will be returned.

See also: **strings.between**, **strings.remove**.

```
strings.isstarting (s, pattern [, true])
```
Determines whether a string `s` is beginning with the substring `pattern`, i.e. whether `pattern` fits entirely to the beginning of the string `s` in case the length of `pattern` is less than that of `s`. The function returns **true** or **false.**

If only two arguments are passed, pattern matching facilities (see Chapter 9.1.3) are supported. If the Boolean constant **true** is passed as a third argument, pattern matching is switched off for faster execution.

If `s` or `pattern` are empty strings or have the same length, the function returns **false**.

The function can be useful in linguistics if you want to check whether a word has a given prefix.

See also: **strings.isending**, **atendof**.

```
strings.isaligned (s)
```
Checks whether the string `s` is aligned on the 4-byte word boundary and returns **true** or **false**.

```
strings.isalpha (s [, t])
```
Checks whether the string `s` consists entirely of alphabetic letters (including diacritics) and returns **true** or **false**.

If the optional string `t` representing a character set is given, the function also checks whether any character in `s` might match one of the characters in `t`.

See also: **strings.isdia**, **strings.isisoalpha**, **strings.islatin**, **strings.ismagic**.

`strings.isalphanumeric (s [, t])`

Checks whether the string `s` consists entirely of numbers or alphabetic letters (including diacritics) and returns **true** or **false**.

If the optional string `t` representing a character set is given, the function also checks whether any character in `s` might match one of the characters in `t`.

See also: **strings.islatinnumeric**.

`strings.isalphaspace (s [, t])`

Checks whether the string `s` consists entirely of alphabetic letters (including diacritics) and/or a white space and returns **true** or **false**.

If the optional string `t` representing a character set is given, the function also checks whether any character in `s` might match one of the characters in `t`.

`strings.isalphaspec (s)`

Checks whether the string `s` consists entirely of the Latin letters a to z, A to Z, or all characters that are not blanks or alphanumeric, and returns **true** or **false**.

See also: **strings.isspec**, **strings.isalphaspace**.

`strings.isascii (s)`

Checks whether the string `s` consists entirely of C unsigned char 7-bit characters only that fits into the UK/US character. It is a direct port to the C function `isascii`, and returns **true** or **false**.

`strings.isblank (s [, true])`

Checks whether the string `s` consists entirely white spaces or tabulators (\t) and returns **true** or **false**. If the option **true** is given, the function checks for tabs, linefeeds, carriage returns, white spaces, vertical tabs, and form feeds.

See also: **strings.isisospace**, **strings.isspace**.

`strings.iscenumeric (s)`

Checks whether the string `s` consists entirely of the digits 0 to 9 and optionally exactly one decimal comma at any position, and returns **true** or **false**.

See also: **strings.isfloat**, **strings.isnumber**, **strings.isnumeric**, **os.setlocale**.

`strings.iscontrol (s)`

Checks whether the string `s` consists entirely of control characters and returns **true** or **false**. Control characters are: '\0', bell, backspace, tab, linefeed, carriage return, and all other characters between ASCII code 0 and 31, plus the DEL key (ASCII code127). The function is the opposite to **strings.isprintable**.

See also: **strings.isblank**, **strings.isprintable**, **strings.isspec**.

`strings.isdia (s)`

Checks whether the string `s` consists entirely of diacritics (such as á, â. ø, Ü) and ligatures (such as ß, Æ) and returns **true** or **false**. The function works correctly with the ISO/IEC 8859-1 character set only.

See also: **strings.isalpha**.

`strings.isending (s, pattern [, true])`

Determines whether a string `s` is ending in the substring `pattern`, i.e. whether `pattern` fits entirely to the end of the string `s` in case the length of `pattern` is less than that of `s`. The function returns **true** or **false.**

If only two arguments are passed, pattern matching facilities (see Chapter 9.1.3) are supported. If the Boolean constant **true** is passed as a third argument, pattern matching is switched off for faster execution.

If `s` or `pattern` are empty strings or both are the same, the function returns **false**.

The function can be useful in linguistics if you want to check whether a word has a given inflectional ending.

See also: **strings.isstarting**, **atendof**.

`strings.isfloat (s)`

Checks whether the string `s` consists entirely of the digits 0 to 9 and exactly one decimal point (or the decimal-point separator at your locale) at any position, and returns **true** or **false**.

See also: **strings.isnumber**, **strings.isnumeric**, **os.setlocale**.

`strings.ishex (s)`

Checks whether the string `s` represents a hexadecimal number which consists of the digits 0 to 9 and or the letters 'a' to 'f' or 'A' to 'F', and returns **true** or **false**.

See also: **strings.isnumber**, **utils.hexlify**.

`strings.isgraph (s)`

Checks whether the string ₛ consists of glyphs only. It is a direct port to the C function `isgraph`, and returns **true** or **false**.

`strings.isisoalpha (s)`

Checks whether the string ₛ consists entirely of ISO 8859/1 Latin-1 alphabetic lower and upper-case characters (including diacritics) and returns **true** or **false**. The function only correctly recognises strings read from a file. Mostly, it cannot process ligatures input in a shell, e.g. the Windows NT or Mac console.

See also: **strings.isalpha**.

`strings.isisolower (s)`

Checks whether the string ₛ consists entirely of ISO 8859/1 Latin-1 alphabetic lower-case characters (including diacritics) and returns **true** or **false**. The function only correctly recognises strings read from a file. Mostly, it cannot process ligatures input in a shell, e.g. the Windows NT or Mac console.

See also: **strings.isalpha**, **strings.isloweralpha**.

`strings.isisoprint (s)`

Checks whether the string ₛ consists entirely of printable ISO 8859/1 Latin-1 letters and returns **true** or **false**.

`strings.isisospace (s)`

Checks whether the string ₛ consists entirely of ISO 8859/1 Latin-1 white spaces and returns **true** or **false**.

See also: **strings.isspace**.

`strings.isisoupper (s)`

Checks whether the string ₛ consists entirely of ISO 8859/1 Latin-1 alphabetic upper-case characters (including diacritics) and returns **true** or **false**. The function only correctly recognises strings read from a file. Mostly, it cannot process ligatures input in a shell, e.g. the Windows NT or Mac console.

See also: **strings.isalpha**, **strings.isupperalpha**.

`strings.islatin (s [, t])`

Checks whether the string ₛ entirely consists of the characters 'a' to 'z', and A' to 'Z'. It returns **true** or **false**. If ₛ is the empty string, the result is always **false**.

If the optional string `t` representing a character set is given, the function also checks whether any character in `s` might match one of the characters in `t`.

See also: **strings.isalpha**.

`strings.islatinnumeric (s [, t])`

Checks whether the string `s` consists entirely of numbers or Latin letters 'a' to 'z' and 'A' to 'Z', and returns **true** or **false**.

If the optional string `t` representing a character set is given, the function also checks whether any character in `s` might match one of the characters in `t`.

See also: **strings.isalphanumeric**.

`strings.isloweralpha (s [, t])`

Checks whether the string `s` consists entirely of the characters a to z and lower-case diacritics, and returns **true** or **false**. If `s` is the empty string, the result is always **false**.

If the optional string `t` representing a character set is given, the function also checks whether any character in `s` might match one of the characters in `t`.

See also: **strings.isisolower**, **strings.isupperalpha**.

`strings.islowerlatin (s [, t])`

Checks whether the string `s` consists entirely of the characters 'a' to 'z', and returns **true** or **false**. If `s` is the empty string, the result is always **false**.

If the optional string `t` representing a character set is given, the function also checks whether any character in `s` might match one of the characters in `t`.

See also: **strings.isupperlatin**.

`strings.ismagic (s)`

Checks whether the string `s` contains one or more magic characters and returns **true** or **false**. In this function, magic characters are anything unlike the letters 'A' to 'Z', 'a' to 'z', and the diacritics listed at the top of this chapter.

See also: **strings.isalpha**.

`strings.ismultibyte (s)`

Detects whether the given string `s` is in UTF-8 encoding and returns two booleans (**true** or **false**): The first Boolean indicates that `s` is compliant to the UTF-8 standard. Remember that a string in ASCII or ISO 8859 encoding is also a valid UTF-8 string.

The second Boolean indicates that ₛ contains at least one multi-byte UTF-8 character, i.e. that at least one character is part of the UTF-8 but not of the ASCII or ISO 8859 standard.

If an integer is returned as a third argument, it will denote the position where the string did not meet UTF-8 criteria.

Please note that the function may not produce correct results with text input in a console. The function can only return correct results if the string to be checked has been read from a file.

See also: **strings.isutf8**, **strings.isisoalpha**.

`strings.isnumber (s)`

Checks whether the string ₛ consists entirely of the digits 0 to 9 and returns **true** or **false**.

See also: **strings.isfloat**, **strings.ishex**, **strings.isnumeric**.

`strings.isnumberspace (s)`

Checks whether the string ₛ consists entirely of the digits 0 to 9 or white spaces and returns **true** or **false**.

`strings.isnumeric (s)`

Checks whether the string ₛ consists entirely of the digits 0 to 9 or digits and optionally exactly one decimal point (or the decimal-point separator at your locale) at any position, and returns **true** or **false**.

See also: **strings.iscenumeric**, **strings.isfloat**, **strings.isnumber**, **os.setlocale**.

`strings.isolower (s)`

Receives an ISO 8859/1 Latin-1 string ₛ and returns a copy of this string with all upper-case letters changed to lower-case. The operator leaves all other characters unchanged.

See also: **lower, strings.isoupper**.

`strings.isoupper (s)`

Receives an ISO 8859/1 Latin-1 string ₛ and returns a copy of this string with all lower-case letters changed to upper-case. The operator leaves all other characters unchanged.

See also: **lower, strings.isoupper**.

`strings.isprintable (s)`

Checks whether the string ₛ consists entirely of characters that can be output at the console (characters with ASCII codes 32 to 255 except the backspace) and returns **true** or **false**. The function is the opposite to **strings.iscontrol**.

`strings.isspace (s)`

Checks whether the string ₛ consists entirely white spaces and returns **true** or **false**.

See also: **strings.isblank**, **strings.isisospace**.

`strings.isspec (s)`

Checks whether the string ₛ consists entirely of punctuation characters (any printing character that is not a white space or alphanumeric), including

white space ¿ ? ¡ ! " # $ @ § % & ' ` * / + − . , ; ( ) [ ] { } | ¦ \ ^ _ ~ = < >

and returns **true** or **false**.

See also: **strings.isalphaspec**, **strings.isspace**, **strings.ismagic**.

`strings.isupperalpha (s [, t])`

Checks whether the string ₛ consists entirely of the capital letters 'A' to 'Z' and upper-case diacritics, and returns **true** or **false**. If ₛ is the empty string, the result is always **false**.

If the optional string ₜ representing a character set is given, the function also checks whether any character in ₛ might match one of the characters in ₜ.

See also: **strings.isisoupper**, **strings.isloweralpha**.

`strings.isupperlatin (s [, t])`

Checks whether the string ₛ consists entirely of the capital letters 'A' to 'Z', and returns **true** or **false**. If ₛ is the empty string, the result is always **false**.

If the optional string ₜ representing a character set is given, the function also checks whether any character in ₛ might match one of the characters in ₜ.

See also: **strings.islowerlatin**.

`strings.isutf8 (s)`

Detects whether the given string contains at least one multibyte and return **true** or **false**.

See also: **strings.ismultibyte**, **strings.isisoalpha**.

`strings.iswrapped (s, t)`

Checks a string `s` with a another string `t` if `t` is missing at the start and the end of `s`, and returns **true** or **false**.

See also: **strings.wrap**, **strings.wrapmissing**.

`strings.iterate (s [, pos [, n]])`

`strings.iterate (s, 0 [, option])`

`strings.iterate (s, delim)`

In the first form, returns an iterator function that, when called returns the next `n` characters in string `str`, starting at position `pos`. `pos` and `n` are 1 by default.

In the second form, when `pos` is zero, returns an iterator function that from the left to right returns each four consecutive characters in string `str` as an unsigned 4-byte integer. The iterator returns Little Endian integers unless the third argument is set to **true** to return Big Endian integers.

In the third form, by passing a string `del` of one or more delimiters as the second argument, returns an iterator function that step-by-step returns a field surrounded by at least one of the delimiters. Pattern-matching is not supported, use **strings.gseparate** instead.

If there are no more characters to process, the factory returns **null**.

See also: **split**, **strings.fields**, **strings.gseparate**, **strings.separate**, **strings.splitfields**, **strings.tobytes**.

`strings.ljustify (s, width [, filler])`

Adds filling characters to the right end of string `s`, as necessary to return a new string of the given `width`. If `s` is a number, it is automatically converted to a string before padding starts. The filling characters may be denoted by the third optional argument `filler` (number or string), otherwise `filler` is a white space by default. If the resulting string is longer than the given `width`, it is truncated to the first `width` characters.

See also: **strings.rjustify**.

`strings.lrtrim (s [, c])`

Returns a new string with all leading and trailing white spaces removed from `s`. If a single character is passed for `c` as an optional second argument, then all leading and trailing characters given by `c` are removed. If `c` is a multi-character string, then if existing it is removed once from the start and once from the end of `s`. The function supports pattern matching.

It does not remove spaces or the given character(s) within the `actual` part of the string.

See also: **trim** operator, **strings.ltrim**, **strings.rtrim**, **strings.wrap**.

`strings.ltrim (s [, c])`

Returns a new string with all leading white spaces removed from `s`. If a single character is passed for `c` as an optional second argument, then all leading characters given by `c` are removed. If `c` is a multi-character string, then if existing it is removed once from the start of `s`. The function supports pattern matching.

See also: **trim** operator, **strings.lrtrim**, **strings.rtrim**.

`strings.match (s, pattern [, init])`

Looks for the first match of `pattern` in the string `s`. If it finds one, then match returns the captures from the pattern; otherwise it returns **null**. If `pattern` specifies no captures, then the whole match will be returned. A third, optional numerical argument `init` specifies where to start the search; its default value is 1 and may be negative.

The function supports pattern matching facilities. For examples and help in case of problems, see Chapter 4.7.7.

See also: **strings.gmatch**, **strings.matches**, **skycrane.xmlmatch**.

`strings.matches (s, pattern [, init])`

Works like **strings.match**, but returns all matches in only one call.

Example:

```
> strings.matches('St. Petersburg, Europe', '([äöüßÄÖÜ%a]*)'):
St              Petersburg              Europe
```

`strings.mfind (s, pattern [, init [, plain]])`

Like **strings.find**, but looks for all the matches of `pattern` in the string `s`. If it finds at least one match, it returns a sequence with at least one pair indicating where the respective match starts and ends, otherwise, it returns **null**.

A third, optional numerical argument `init` specifies where to start the search; its default value is 1 and may be negative. A value of **true** as a fourth, optional argument `plain` turns off the pattern matching facilities (see Chapter 9.1.3), so the function does a plain `find substring` operation, with no characters in `pattern` being considered `magic`. Note that if `plain` is given, then `init` must be given as well.

Contrary to **strings.find**, if the pattern has captures, then in a successful match the captured values are not returned.

See also: **in**, **atendof**, and **instr**, **strings.find**, **strings.matches**.

strings.pack (fmt, v1, v2, ···)

Returns a binary string containing the values `v1`, `v2`, etc. serialized in binary form (packed) according to the format string `fmt`, see Chapter 9.1.4.

strings.packsize (fmt)

Returns the size of a string resulting from **strings.pack** with the given format. The format string cannot have the variable-length options `'s'` or `'z'`. For format strings, see Chapter 9.1.4.

strings.random (length [, kind [, l [, u]]])

Creates a random string of the given fixed `length`. By default, i.e. `kind` is set to `'base64'`, a Base64 string consisting of the characters

    ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/

will be returned. If the second argument kind is `'ascii'`, a random ASCII string consisting of characters in the range ASCII 32 to ASCII 126 will be returned. You can change the upper and lower bounds by explicitly passing the non-negative integers `l` and `u`.

strings.remove (s, pos [, len])
strings.remove (s, p [, n])
strings.remove (s, p [, ···])

The function removes a substring from a string. It supports pattern matching.

In the first form, starting from string position `pos`, the function removes `len` characters from string `s`. The return is a new string. If `len` is not given, it defaults to one character to be deleted.

It is not an error if `len` is greater than the actual length of `s`. In this case all characters starting at position `pos` are deleted.

In the second form, substring `p` is removed `n` times from string `s`. The default for `n` is **infinity**, i.e. all occurrences of `p` are removed.

In the third form, one or more substrings `p`, ... are removed from string `s`, in the order of the arguments.

See also: **replace**, **strings.include**.

strings.repeat (s, n [, delimiter])

Returns a string that is the concatenation of `n` copies of the string `s`. An optional `delimiter` string may also be given.

strings.reverse (s)

Returns a string that is the string `s` reversed. See also: **reverse**, **stack.reversed**.

strings.rjustify (s, width [, filler])

Adds filling characters to the beginning of string `s`, as necessary to return a new string of the given `width`. If `s` is a number, it is automatically converted to a string before padding begins. The filling characters may be denoted by the third optional argument `filler` (number or string), otherwise `filler` is a white space by default. If the resulting string is longer than the given `width`, it is truncated to the last `width` characters.

See also: **strings.ljustify**.

strings.rtrim (s [, c])

Returns a new string with all trailing white spaces removed from `s`. If a single character is passed for `c` as an optional second argument, then all trailing characters given by `c` are removed. If `c` is a multi-character string, then if existing it is removed once from the end of `s`. The function supports pattern matching.

See also: **trim** operator, **strings.lrtrim**, **strings.ltrim**.

strings.rotateleft (s, n [, xorkey [, xorval]])

Rotates all the bits in the string `s` `n` bits to the left, with `n` in range 0 .. 7.

The `n` bits dropping off the beginning of the string will be appended to the resulting string, so that there is no information loss when calling **strings.rotateright** to decrypt it.

For optional arguments `xorkey` and `xorval` see **strings.rotateright**.

`strings.rotateright (s, n [, xorkey [, xorval]])`

Rotates all the bits in the string `s` `n` bits to the right, with `n` in range 0 .. 7.

The `n` bits dropping off the end of the string will be prepended to the resulting string, so that there is no information loss when calling **strings.rotateleft** to decrypt it.

You can optionally xor the string by passing the third argument `xorkey`, an integer in the range 0 .. 255. (Note that in case the string might be corrupted, the function issues an error.) By explicitly setting the optional fourth argument `xorval` to **true**, you can achieve further obfuscation of the string while xoring.

`strings.separate (s, d [, any])`

Splits a string `s` into its tokens. `d` is a string that specifies a set of delimiters that may surround the token to be extracted. Thus, the delimiter in front of a token may be different from the delimiter at its end. All the tokens or returned in a sequence in sequential order. If `s` only consists of characters that are part of `d`, or if `s` or `d` are empty strings, the function returns **fail**.

```
> strings.separate('a word, another word.', ' .,'):
seq(a, word, another, word)
```

If `any` third argument is passed, then a) the function returns a sequence with one empty string if `s` is the empty string instead of **fail**, and b) if none of the delimiters could be found in `s`, returns a sequence with `s` in it instead of **fail**.

See also: **split** operator, **strings.iterate**, **strings.gseparate**, **strings.splitfields**.

`strings.shannon (s)`

Returns the normalised specific Shannon entropy, the specific Shannon entropy, and the total information entropy (in bits) for string `s`, in this order.

The function does not look for any patterns that might be available for compression, so its use is quite limited and **gzip.deflate** might be a better alternative.

`strings.splitfields (s, wrapper, delim)`

Splits a string `s` containing fields delimited by substring `delim` into its fields and returns them in a sequence. The function is suited to parse database or other dumps.

One or more fields may each be wrapped by substring `wrapper` - if the wrapper exist, the respective leading and trailing substring `wrapper` is removed from each individual field. Pass the empty string for `wrapper` if no wrappers are to be deleted.

See also: **split** operator, **strings.fields**, **strings.iterate**, **strings.gseparate**, **strings.separate**.

strings.strchr (s, i)

The function is an interface to the C strchr function, searches `s` for a single character represented by its ASCII code `i` and returns a substring starting from the first match to the end of `s`. The second return is the position of the match, starting from 1. It returns **null** and 0 if no match was found and issues an error if needle is non-positive.

See also: **abs**, **strings.strrchr**, **strings.strstr**.

strings.strrchr (s, i)

The function is an interface to the C strrchr function, searches `s` backwards from the end for a single character represented by its ASCII code `i` and returns a substring starting from the first match to the end of `s`. The second return is the position of the match, starting from 1. It returns **null** and 0 if no match was found and issues an error if needle is non-positive.

See also: **abs**, **strings.strchr**, **strings.strstr**.

strings.strcmp (s1, s2)

The function calls the C function strcmp and returns its result, "a value that has the same sign as the difference between the first differing pair of characters" (GNU C Library manual).

See also: **strings.compare**, **strings.strverscmp**.

strings.strlen (s)

Returns the length of string `s`: the first return is the result of the call to the internal C function strlen, and the second return is the internally stored length of `s`, returned by Agena's **size** operator.

The difference between strlen and **size** is that C's strlen only counts the number of characters up to and excluding the first embedded zero (i.e. character '\0'), whereas **size** returns the real length including embedded zeros, but without the terminating zero.

Example:

```
> s := 'abc' & char(0) & 'defgh';
> # 3 chars up to the first embedded zero, 9 chars at all

> strings.strlen(s):
3       9
```

`strings.strstr (s1, s2)`

The function is an interface to the C strstr function, searches `s1` for a substring `s2` and returns a substring starting from the first match to the end of `s1`. The second return is the position of the match, starting from 1. It returns **null** and 0 if no match was found. If `s2` is an empty string, the function returns `s1`.

See also: **strings.strchr**, **strings.strrchr**.

`strings.strverscmp (s1, s2)`

The function compares two version strings. It is a direct interface to the GNU C strverscmp function. The following is a summary of the GNU documentation:

"If you have files jan1, jan2, ..., jan9, jan10, ..., it feels wrong when an application orders them jan1, jan10, ..., jan2, ..., jan9, because the expected order is just: jan1, jan2, ..., jan9, jan10, ...

The function returns an integer less than, equal to, or greater than zero if s1 is found, respectively, to be earlier than, equal to, or later than s2.

Both input strings should be in plain ASCII."

See also: **strings.compare**.

`strings.tobytes (s [, option [, bigendian]])`

Converts a string `s` into a sequence of its numeric ASCII codes. If the string is empty, an empty sequence will be returned. If `option` is **true** or the integer 4, the function returns word-aligned 4-byte unsigned integers instead of individual bytes. If `bigendian` is **true**, then with word-aligned 4-byte unsigned integers the result is in Big Endian notation, otherwise it is Little Endian.

Note that numerical codes are not necessarily portable across platforms.

Example:

```
> s := strings.tobytes('agena', 4):  # convert to 4-byte integers
seq(1852139361, 97)

> str := '';

> for i in s do  # convert each 4-byte integer
>    t := bytes.tobytes(i, 4)   # to four single bytes and convert
>    str &:= strings.tochars(t) # back to string
> od
> str:
agena

> # or just simply:
```

```
> strings.tochars(s, 4):
```

See also: **strings.iterate**, **bytes.tobytes**, **strings.tochars**, **utils.hexlify**.

strings.tochars (···)

strings.tochars (s [, nbytes [, little]])

In the first form, receives zero or more integers in the range 0 .. 255 and returns a string with length equal to the number of arguments, in which each character has the internal numerical code equal to its corresponding argument.

In the second form, converts all the integers in sequence `s` to a string. By default, `s` is assumed to contain integers in the range 0 .. 255. If `nbytes` is 4, `s` should include unsigned 4-byte integers. If `little` is the Boolean **true** - the default - the integers are converted to Little Endian before assembling the string, otherwise pass **false**.

Note that numerical codes are not necessarily portable across platforms.

See also: **strings.tobytes**.

strings.tolatin (s)

Creates a dynamically allocated copy of string `s`, changing the encoding from UTF-8 to ISO-8859-15. Unknown code points are returned unchanged. The return is a string. ISO-8859-15 is ISO-8859-1 plus the Euro symbol.

See also: **aconv** package, **strings.toutf8**.

strings.tolower (s [, option])

Converts all uppercase letters in string `s` to lowercase. By default, only the characters 'A' to 'Z' are transformed. If you pass any `option`, then also the diacritics listed at the end of Chapter 9.1 are converted, too.

See also: **lower**, **strings.toupper**.

strings.toupper (s [, option])

Converts all lowercase letters in string `s` to uppercase. By default, only the characters 'a' to 'z' are transformed. If you pass any `option`, then also the diacritics listed at the end of Chapter 9.1 are converted, too.

See also: **lower**, **strings.toupper**.

`strings.toutf8 (s)`

Creates a dynamically allocated copy of string `s`, changing the encoding from ISO-8859-15 to UTF-8. The return is a string. ISO-8859-15 is ISO-8859-1 plus the Euro symbol.

See also: **aconv** package, **strings.isutf8**, **strings.tolatin**, **strings.utf8size**.

`strings.transform (f, s)`

Applies a function `f` to the ASCII value of each character in string `s` and returns a new string. `f` must return an integer in the range [0, 255], otherwise an error will be issued.

Note that numerical codes are not necessarily portable across platforms.

`strings.uncapitalise (s)`

Converts the first character in string `s` to lower case - if possible - and returns the uncapitalised string. If `s` is the empty string, it is simply returned. It also converts ligatures if the Western European character set is being used.

See also: **lower**, **strings.capitalise**.

`strings.unpack (fmt, s [, pos]])`

Returns the values packed in string `s` (see **strings.pack**) according to the format string `fmt`, see Chapter 9.1.4. An optional `pos` marks where to start reading in `s` (default is 1). After the read values, this function also returns the index of the first unread byte in `s`. If `s` - depending on the requested transformation - is too short, the function just returns **null** and zero.

`strings.utf8size (s)`

Determines the size of the string `s` in UTF-8 encoding and returns a non-negative integer. The return is not the number of bytes used to represent a UTF-8 string, but the number of single- and multi-byte `UTF-8 characters`. Thus, for example, while `size  strings.toutf8('à')` returns 2, `strings.utf8size(strings.toutf8('à'))` returns 1.

Please note that the function may not produce correct results with text input in a console. The function can only return correct results if the string to be checked has been read from a file.

See also: **size**, **strings.isutf8**.

`strings.words (s [, delim [, true]])`

Counts the number of words in a string `s`. A word is any sequence of characters surrounded by white spaces or its left and/or right borders. The user can define any other delimiter by passing an optional character `delim` (of type string) as a second argument. If the third argument is **true**, then succeeding delimiters are ignored. The return is a number.

See also: **strings.hits**.

`strings.wrap (s, t)`

Wraps a string `s` with another strings `t`, returning the Agena equivalent of `t` & `s` & `t`.

See also: **strings.iswrapped**, **strings.lrtrim**, **strings.wrapmissing**.

`strings.wrapmissing (s, t)`

Wraps a string `s` with string `t` if `t` is missing at the start and the end of `s`; otherwise simply returns `s`.

See also: **strings.iswrapped**, **strings.wrap**.

9.1.3 Patterns

Character Class:

A character class is used to represent a set of characters. The following combinations are allowed in describing a character class:

- **x:** (where x is not one of the magic characters `^$()%.[]*+-?`) represents the character x itself.
- **.:** (a dot) represents all characters.
- **%a:** represents all letters.
- **%c:** represents all control characters.
- **%d:** represents all digits.
- **%l:** represents all lowercase letters.
- **%k:** represents all upper and lower-case consonants, y and Y are not considered consonants.
- **%p:** represents all punctuation characters.
- **%s:** represents all space characters, e.g. white spaces, newlines, tabulators, and carriage returns,
- **%u:** represents all uppercase letters.
- **%v**: represents all upper and lower-case vowels including the letters y and Y.
- **%w:** represents all alphanumeric characters.
- **%x:** represents all hexadecimal digits.
- **%z:** represents the character with representation 0.
- **%<y>:** (where <y> is any non-alphanumeric character) represents the character y. This is the standard way to escape the magic characters. Any punctuation character (even the non magic) can be preceded by a `'%'` when used to represent itself in a pattern.
- **[set]:** represents the class which is the union of all characters in *set.* A range of characters may be specified by separating the end characters of the range with a `'-'`. All classes %y described above may also be used as components in set. All other characters in set represent themselves. For example, [%w_] (or [_%w]) represents all alphanumeric characters plus the underscore, [0-7] represents the octal digits, and [0-7%l%-] represents the octal digits plus the lowercase letters plus the `'-'` character.
- The interaction between ranges and classes is not defined. Therefore, patterns like [%a-z] or [a-%%] have no meaning.
- **[^set]:** represents the complement of *set*, where set is interpreted as above.

For all classes represented by single letters (%a, %c, %v etc.), the corresponding uppercase letter represents the complement of the class. For instance, %S represents all non-space characters.

The definitions of letter, space, and other character groups depend on the current locale. In particular, the class [a-z] may not be equivalent to %l.

Pattern Item:

A *pattern item* may be

- a single character class, which matches any single character in the class;
- a single character class followed by '*', which matches 0 or more repetitions of characters in the class. These repetition items will always match the longest possible sequence;
- a single character class followed by '+', which matches 1 or more repetitions of characters in the class. These repetition items will always match the longest possible sequence;
- a single character class followed by '-', which also matches 0 or more repetitions of characters in the class. Unlike '*', these repetition items will always match the *shortest possible sequence;*
- a single character class followed by '?', which matches 0 or 1 occurrence of a character in the class;
- %n, for n between 1 and 9; such item matches a substring equal to the n-th captured string (see below);
- %b*xy*, where x and y are two distinct characters; such item matches strings that start with x, end in y, and where the x and y are balanced. This means that, if one reads the string from left to right, counting +1 for an x and -1 for a y, the ending y is the first y where the count reaches 0. For instance, the item %b() matches expressions with balanced parentheses;
- %f[set], a frontier pattern; such item matches an empty string at any position such that the next character belongs to set and the previous character does not belong to set. The set `set` is interpreted as previously described. The beginning and the end of the subject are handled as if they were the character '\0'.

Pattern:

A *pattern* is a sequence of pattern items. *A* '^' at the beginning of a pattern anchors the match at the beginning of the subject string. A '$' at the end of a pattern anchors the match at the end of the subject string. At other positions, '^' and '$' have no special meaning and represent themselves.

Captures:

A pattern may contain sub-patterns enclosed in parentheses; they describe captures. When a match succeeds, the substrings of the subject string that match captures are stored (captured) for future use. Captures are numbered according to their left parentheses. For instance, in the pattern '(a*(.)%w(%s*))', the part of the string matching 'a*(.)%w(%s*)' is stored as the first capture (and therefore has number 1); the character matching '.' is captured with number 2, and the part matching '%s*' has number 3.

As a special case, the empty capture () captures the current string position (a number). For instance, if we apply the pattern '()aa()' on the string 'flaaap', there will be two captures: 3 and 5.

A pattern cannot contain embedded zeros. Use `%z` instead.

9.1.4 Format Strings for Pack and Unpack

The first argument to **strings.pack**, **strings.packsize**, and **strings.unpack** is a format string, which describes the layout of the structure being created or read.

A format string is a sequence of conversion options. The conversion options are as follows:

- `<`: sets Little Endian
- `>`: sets Big Endian
- `=`: sets native Endian
- `![n]`: sets maximum alignment to n (default is native alignment)
- `b`: a signed byte (char)
- `B`: an unsigned byte (char)
- `h`: a signed short (native size)
- `H`: an unsigned short (native size)
- `l`: a signed long (native size)
- `L`: an unsigned long (native size)
- `j`: a lua_Integer
- `J`: a lua_Unsigned
- `T`: a size_t (native size)
- `i[n]`: a signed int with n bytes (default is native size)
- `I[n]`: an unsigned int with n bytes (default is native size)
- `f`: a float (native size)
- `d`: a double (native size)
- `n`: an Agena number of C type double = lua_Number
- `cn`: a fixed-sized string with n bytes
- `z`: a zero-terminated string
- `s[n]`: a string preceded by its length coded as an unsigned integer with n bytes (default is a size_t)
- `x`: one byte of padding
- `Xop`: an empty item that aligns according to option op (which is otherwise ignored)
- `' '`: (a white space) ignored

(A "[n]" means an optional integral numeral.) Except for padding, spaces, and configurations (options "xX <=>!"), each option corresponds to an argument in **strings.pack** or a result in **strings.unpack**.

For options "!n", "sn", "in", and "ln", n can be any integer between 1 and 16. All integral options check overflows; **strings.pack** checks whether the given value fits in

the given size; **strings.unpack** checks whether the read value fits in a Lua integer. For the unsigned options, Lua integers are treated as unsigned values too.

Any format string starts as if prefixed by "!1=", that is, with maximum alignment of 1 (no alignment) and native endianness.

Native endianness assumes that the whole system is either Big or Little Endian. The packing functions will not emulate correctly the behavior of mixed-Endian formats.

Alignment works as follows: For each option, the format gets extra padding until the data starts at an offset that is a multiple of the minimum between the option size and the maximum alignment; this minimum must be a power of 2. Options "c" and "z" are not aligned; option "s" follows the alignment of its starting integer.

All padding is filled with zeros by **strings.pack** and ignored by **strings.unpack**.

9.2 memfile - Memory File for Strings

The **memfile** library implements a character buffer, i.e. a `memory file` userdata that stores a string of almost unlimited length, along with functions to administer it. It is useful if you have to iteratively concatenate a lot of strings, being 20 times faster than the **&** operator.

Typical usage:

```
> m := memfile.charbuf()            # create a memory file

> memfile.append(m, 'nasa', 'jpl') # put two strings into it

> f := memfile.iterate(m, 1, 2) # from position 1, return 2 chars per call

> f():
na

> f():
sa

> f():
jp

> f():
l

> f():
null
```

Let us declare a bit field of two bytes:

```
> b := memfile.bitfield(16)
```

and in these two bytes, set the even bits to 1, i.e. twice to 0b10101010 = 170:

```
> for i to 16 do if even(i) then b[i] := 1 fi od
```

The contents of the field is:

```
> b:
bitfield(0b10101010, 0b10101010)
```

Get some bits, the first and the tenth:

```
> b[1]:
0
> b[10]:
1
```

Clear the bits in the first byte:

```
> for i to 8 do if even(i) then b[i] := 0 fi od
```

```
> b:
bitfield(0b00000000, 0b10101010)
```

The package provides the following metamethods:

Metamethod	Functionality
`'__index'`	read operation, e.g. n[p] or n[p to q], with p, q indices, both counting from 1; with bit fields, reads a bit, not a byte
`'__writeindex'`	write operation, e.g. n[p] := value, with p the index, counting from 1; with bit fields, sets a bit, not a byte
`'__size'`	**size** operator, number of characters currently stored; with bit fields returns the number of bits in the field, not bytes
`'__in'`	**in** operator
`'__notin'`	**notin** operator
`'__eq'`	= equality operator
`'__empty'`	**empty** operator
`'__filled'`	**filled** operator
`'__tostring'`	formatting for output at the console; with bit fields, returns binary representations
`'__gc'`	garbage collection

The functions are:

memfile.append (memfile, v [, ···] [, delim=str])

Appends one or more numbers, strings, Booleans or **null**'s `v`, etc. to the end of `memfile`. The function returns nothing.

You can specify the optional character delimiter option `delim = str` that separates each value to be added to the memory file, e.g. `memfile.append(m, 'a', 'b', 'c', delim = ';')` actually adds the string `'a;b;c;'`. You may later on drop the final delimiter by calling **memfile.dump** with the size of the delimiter.

See also: **memfile.move**, **memfile.rewind**.

memfile.attrib (memfile)

Returns the total capacity of a `memfile` and the current number of allocated bytes, in this order.

See also: **memfile.getsize**.

`memfile.bitfield (n, [,···])`

Creates a bit field of at least `n` bits and optionally sets zeros or ones into this field. The return is a byte buffer with initially all positions set to zero.

If you pass optional ones or zeros, or the Booleans **true** or **false**, they are set from the right end to the left end of the new bit field, e.g. if we have

```
> b := memfile.bitfield(4, 1, 1, 1, 0)
```

we will store 0b0111 = 7 decimal into the field. If you need it the other way around, execute something like

```
> b := memfile.bitfield(4, unpack(reverse(reg(1, 1, 1, 0))))
```

The number of bits actually allocated is always a multiple of 8, i.e. the field is filled up to whole bytes. In the example above, instead of four bits we created a bit field of eight bits.

Since the memory file created is no different from the one created by **memfile.charbuf**, with the exception of the metatable, you can apply all the other **memfile** functions on it. The bit field metatable `bitfield` contains specialised functions to get, set, determine the size and print bit fields, which work on the bit and not byte level.

See also: **memfile.attrib**, **memfile.bytebuf**, **memfile.charbuf**, **memfile.resize**.

`memfile.bytebuf (n, [,···])`

Creates a memory file of fixed size `n` bytes and fills it with zeros by default. It can also initialise the buffer with bytes (second to last argument).

Since the memory file created is no different from the one created by **memfile.charbuf**, you can apply all the other **memfile** functions on it.

See also: **memfile.attrib**, **memfile.bitfield**, **memfile.charbuf**, **memfile.resize**.

`memfile.charbuf ([v [,···] [, delim=str]])`

Creates a memory file and optionally puts one or more numbers, strings, Booleans or **null**'s `v` into it. The function returns the memory file created.

You can specify the optional character delimiter option `delim = str` that separates each value to be added to the memory file, e.g. `memfile.charbuf('a', 'b', 'c', delim = ';')` actually adds the string `'a;b;c;'`. You may later on drop the final delimiter by calling **memfile.dump** with the size of the delimiter.

See also: **memfile.attrib**, **memfile.bytebuf**, **memfile.resize**.

`memfile.clearbit (memfile, n)`

`memfile.clearbit (memfile, pos, n)`

In the first form, unsets absolute bit position `n` in the `memfile`, i.e. sets it to 0. `n` counts from 1. To set a bit, use **memfile.setbit**.

In the second form, in byte no. `pos` of the `memfile`, unsets the `n`-th bit, i.e. sets it to 0, where `n` > 0.

The return is the modified byte at byte position `pos`.

See also: **memfile.getbit**, **memfile.setbit**, **memfile.setbyte**, **memfile.setchar**, **memfile.setfield**, **memfile.setitem**.

`memfile.dump (memfile [, n])`

Without a second argument, returns the whole string stored in `memfile` and resets `memfile` completely to its original state, so that it can store a new string. With a byte buffer, the function refills it with zeros again after dumping the contents.

If a positive integer `n` is passed as an optional argument, then the function just dumps `n` bytes from the end (tail) of the memory file and returns them as a string, leaving the rest of the `memfile` untouched. If the `memfile` should be empty after this operation, it is reset, which is equal to calling the function without an optional argument.

See also: **memfile.get**, **memfile.getbytes**.

`memfile.find (memfile, str [, pos])`

`memfile.find (memfile, byte [, pos])`

With a character buffer, searches `memfile` for a substring `str` and returns its position, an integer, or **null** if the string has not been found. The optional argument `pos` indicates the position where to start the search, and is 1 by default.

With a byte buffer, searches for the given `byte`, a non-negative integer in the range 0 to 255.

See also: **memfile.substring**, **in** metamethod.

`memfile.get (memfile [, n])`

Without a second argument, returns the whole string stored in `memfile`.

If a positive integer `n` is passed as an optional argument, then the function just returns `n` characters from the end (tail) of the memory file.

The function contrary to **memfile.dump** does not remove any contents and also does not reset or re-size the memory file.

See also: **memfile.dump**, **memfile.getbytes**, **memfile.substring**.

```
memfile.getbit (memfile, n)
```
```
memfile.getbit (memfile, p [, n])
```

In the first form, returns the bit stored at absolute bit position `n` in the `memfile`. `n` counts from 1.

In the second form, from byte no. `p` in the `memfile`, returns the `n`-th bit, where `p` and `n` > 0.

The return is either 1 or 0.

See also: **memfile.clearbit**, **memfile.getfield**, **memfile.setbit**.

```
memfile.getbyte (memfile, pos [, option])
```

From `memfile`, returns the byte at position `pos`, with `pos` <> 0. If `pos` is negative, the position is relative to the end of the string. The return is an integer in the range [0, 255].

If any `option` is given, then the function returns a string with the binary representation of the byte at `pos`, e.g. `'0b10000000'`.

See also: **memfile.getbit**, **memfile.getbytes**, **memfile.getchar**, **memfile.getitem**.

```
memfile.getbytes (memfile [, pos])
```

From `memfile`, starting from byte position `pos` in `memfile`, returns a register of all the bytes (integers in the range [0, 255]) stored in the memory file. `pos` is 1 by default, i.e. all bytes will be returned. If `pos` is negative, the position is relative to the end of the string.

See also: **memfile.dump**, **memfile.getbit**, **memfile.getbyte**, **memfile.getchar**, **memfile.getitem**, **memfile.getsize**, **size** metamethod.

```
memfile.getchar (memfile, pos)
```

From `memfile`, returns the character at position `pos`, with `p`<> 0. If `pos` is negative, the position is relative to the end of the string. The return is a string of size 1: the character.

See also: **memfile.getbyte**, **memfile.get**, **memfile.getitem**, **memfile.substring**.

`memfile.getfield (memfile, n)`

Returns the bit stored at absolute bit position `n` in `memfile`. `n` counts from 1.

The return is either 1 or 0.

See also: **memfile.setfield**, **memfile.clearbit**, **memfile.getbit**, **memfile.setbit**.

`memfile.getitem (memfile, p [, n])`

From `memfile`, returns the substring starting at position `p` and of length `n`, with non-zero `p`. If `p` is negative, the position is relative to the end of the string. By default, `n` is 1.

See also: **memfile.getbyte**, **memfile.getchar**, **memfile.getsize**, **memfile.iterate**, **memfile.setitem**, **memfile.substring**.

`memfile.getsize (memfile)`

From `memfile`, returns the number of characters (bytes) stored in it.

See also: **size** metamethod, **memfile.attrib**.

`memfile.iterate (memfile [, pos [, n]])`

Returns an iterator function that when called, returns the next `n` characters stored in `memfile`, starting at position `pos`. If there are no more characters, the factory returns **null**. By default `pos` and `n` are 1.

See also: **memfile.getitem**, **memfile.substring**.

`memfile.map (f, memfile [,···])`

Maps a function `f` on each character in `memfile`, in-place. `f` must always return a number or a string.

`memfile.move (memfile, pos)`

Moves the end of the current `memfile` to the given position `pos`, inclusive, with `pos` a non-negative integer, or in other words: changes the size of the `memfile` without reallocating memory. If `pos` is 0, the `memfile` is cleared. If `pos` is, for example 2, then if you call **memfile.append** thereafter with a substring, it will be added starting at position 3, preserving the values at position 1 and 2. The function returns nothing.

See also: **memfile.append**, **memfile.resize**, **memfile.rewind**.

`memfile.read (fh, memfile [, bufsize])`

Reads data from the file denoted by its filehandle `fh` into the given `memfile` userdata.

The file should have previously been opened with **binio.open** and should finally be closed with **binio.close**.

By default, the function reads in the entire file if `bufsize` is not given.

If a positive integer has been passed with `bufsize`, the function only reads in the given number of bytes per each call, returns the number of bytes actually read and increments the file position thereafter so that the next bytes in the file can be read with a new call to **memfile.read**.

(Passing the `bufsize` argument may also be necessary if your platform requires that an internal input buffer is aligned to a certain block size.)

If the end of the file has been reached, or there is nothing to read at all, **null** will be returned. In case of an error, it quits with the respective error.

Example:

```
> m := memfile.charbuf();

> print(size m);  # should be zero

> fd := binio.open('memfile.bin');
> pos := 1;

> # the following loop is equivalent to the simple call
> # `memfile.read(fd, m)`:

> do
>    pos := memfile.read(fd, m, 512)  # read 512 bytes per each call
> until pos = null;

> binio.close(fd);  # should now be non-zero

> print(size m);
```

See also: **memfile.write**.

`memfile.resize (memfile, n [, flag])`

Resizes the `memfile` to exactly `n` places (bytes), with `n > 0`. It can grow or shrink a memory file and in the latter case preserves the remaining content. If the memory file is to be enlarged, the function optionally fills the new space with zeros if the third argument `flag` **true** is given, otherwise you may just pass **false** which is the default.

You may call **bytes.optsize** before to determine the optimal number of places (bytes) in the memory file to be word-aligned.

See also: **memfile.attrib**, **memfile.bytebuf**, **memfile.charbuf**, **memfile.getitem**, **memfile.move**, **memfile.rewind**.

`memfile.reverse (memfile)`

Reverses the data, i.e. bytes, in a memory file, in-place. The function returns nothing.

`memfile.rewind (memfile)`

Sets the current size of a `memfile` to zero, effectively clearing the buffer without re-allocating memory. The function returns nothing.

See also **memfile.append**, **memfile.move**.

`memfile.setbit (memfile, n)`
`memfile.setbit (memfile, pos, n [, val])`

In the first form, sets absolute bit position `n` in the `memfile` to 1. `n` counts from 1. To clear a bit, use **memfile.clearbit**.

In the second form, in byte no. `pos` of the `memfile`, sets the `n`-th bit to `val`, where `val` is either a Boolean or 0 or 1 and `n` > 0. If `val` is omitted, sets the bit to 1.

The return is the modified byte at byte position `pos`.

See also: **memfile.clearbit**, **memfile.getbit**, **memfile.setbyte**, **memfile.setchar**, **memfile.setfield**, **memfile.setitem**.

`memfile.setbyte (memfile, pos, i [, count])`

Sets byte `i` of type (non-negative) integer into `memfile` at the *existing* position pos, with `pos` <> 0.

If `pos` is negative, the position is relative to the end of the string. `i` should be an integer in the range [0, 255]. The function returns nothing.

If a fourth argument `count` is given, then the function sets `count` bytes - starting from position `pos` - to the given byte. By default, `count` is 1. If `pos` has not been set before, the function fills all preceding positions with zeros, if they have not yet been already set to any byte.

See also: **memfile.getbyte**, **memfile.getchar**, **memfile.setbit**, **memfile.setchar**, **memfile.setitem**.

`memfile.setchar (memfile, pos, c)`

Sets character `c` of type string into `memfile` at the *existing* position `pos`, with `pos <>` 0. If `c` is longer than one character only the first character is written to the memory file.

If `pos` is negative, the position is relative to the end of the string. `i` should be an integer in the range [0, 255]. The function returns nothing.

If a fourth argument `count` is given, then the function sets `count` places - starting from position `pos` - to the given character. By default, `count` is 1. If `pos` has not been set before, the function fills all preceding positions with white spaces, if they have not yet been already set.

See also: **memfile.getbyte**, **memfile.getchar**, **memfile.setbit**, **memfile.setchar**, **memfile.setitem**.

`memfile.setfield (memfile, n, val)`

Sets the `n`-th bit in a memory file to `val`, where `val` is either the Boolean **true** or **false**, or 0 or 1. In `n` is negative, then the |`n`|-th bit from the left side of the buffer is set or unset.

The return is the modified byte in which the bit resides.

See also: **memfile.clearbit**, **memfile.getfield**, **memfile.setbit**.

`memfile.setitem (memfile, str, pos [, count])`
`memfile.setitem (memfile, byte, pos [, count])`

In the first form, sets a substring `str` into `memfile` at position `pos`, with non-zero `pos`. If the substring is too long, the function issues an error. If `pos` is negative, the position is relative to the end of the string.

In the second form, does the same with the non-negative integer `byte`.

The function returns nothing.

See also: **memfile.getitem**, **memfile.setbyte**, **memfile.setchar**.

`memfile.shift (memfile, n)`

Rotates the contents of the buffer `memfile` `n` bytes to the right if `n` is positive, and `n` bytes to the left if `n` is negative. The function returns nothing.

See also: **memfile.move**, **memfile.substring**.

`memfile.substring (memfile, p [, q])`

From `memfile`, returns the substring from position `p` to position `q`, with non-zero `p`, `q`. If `p` or `q` are negative, the respective positions are relative to the end of the string. `q` is `p` by default.

See also: **memfile.get**, **memfile.getitem**.

`memfile.write (fh, memfile [, pos [, nchars]])`

Writes the string in a `memfile` userdata to the file denoted by its numeric file handle `fh`.

The file should be opened with **binio.open** and closed with **binio.close** after completion.

The start position `pos` is 1 by default but can be changed to any other valid position in the `memfile`.

The number of characters (not necessarily bytes) to be written can be changed by passing an optional fourth argument `nchars`, a positive number, and by default equals the total number of characters in `memfile`. (Passing the `nchars` argument may also be necessary if your platform requires that buffers must be aligned to a particular block size.)

The function returns the index of the next start position (an integer) for a further call to **memfile.write** to write further characters, where the return should be passed to the third `pos` argument.

If the end of the string in `memfile` has been reached, the function returns **null** and flushes all unwritten content to the file so that you do not have to call **binio.sync** manually.

No further information is stored to the file created.

Example on how to write a string of 8,000 characters piece-by-piece:

```
> m := memfile.charbuf();

> to 1000 do
>    memfile.append(m, 'nasa/jpl')
> od;

> fd := binio.open('memfile.bin');

> pos := 1;

> # The following is equivalent to "memfile.write(fd, m)":

> do  # write 1024 values per each call
>    pos := memfile.write(fd, m, pos, 1024)
> until pos = null;
```

```
> binio.close(fd);
```

Use **binio.sync** if you want to make sure that any unwritten content is written to the file when calling **memfile.write** multiple times.

See also: **memfile.read**.

9.3 utf8 - UTF-8 Helpers

As a *plus* package, the **utf8** package is not part of the standard distribution and must be activated with the **import** statement, i.e. `import utf8`.

This library provides basic support for UTF-8 encoding. It provides all its functions inside the table utf8. This library does not provide any support for Unicode other than the handling of the encoding. Any operation that needs the meaning of a character, such as character classification, is outside its scope.

Unless stated otherwise, all functions that expect a byte position as a parameter assume that the given position is either the start of a byte sequence or one plus the length of the subject string. As with many string functions, negative indices count from the end of the string.

Functions that create byte sequences accept all values up to 0x7FFFFFFF, as defined in the original UTF-8 specification; that implies byte sequences of up to six bytes.

Functions that interpret byte sequences only accept valid sequences (well formed and not overlong). By default, they only accept byte sequences that result in valid Unicode code points, rejecting values greater than 10FFFF and surrogates. A boolean argument **lax**, when available, lifts these checks, so that all values up to 0x7FFFFFFF are accepted. (Not well formed and overlong sequences are still rejected.)

utf8.chars (···)

Receives zero or more integers, converts each one to its corresponding UTF-8 byte sequence and returns a string with the concatenation of all these sequences.

utf8.charpattern

The pattern (a string, not a function) "[\0-\x7F\xC2-\xF4][\x80-\xBF]*", which matches exactly one UTF-8 byte sequence, assuming that the subject is a valid UTF-8 string.

utf8.codes (s [, lax])

Returns values so that the construction

```
for p, c in utf8.codes(s) do body od
```

will iterate over all characters in string `s`, with `p` being the position (in bytes) and `c` the code point of each character. It raises an error if it meets any invalid byte sequence.

`utf8.codepoint (s, [, i [, j [, lax]]])`

Returns the codepoints (as integers) from all characters in `s` that start between byte position `i` and `j` (both included). The default for `i` is 1 and for `j` is i. It raises an error if it meets any invalid byte sequence.

`utf8.len (s, [, i [, j [, lax]]])`

Returns the number of UTF-8 characters in string `s` that start between positions i and j (both inclusive). The default for `i` is 1 and for `j` is -1. If it finds any invalid byte sequence, returns a false value plus the position of the first invalid byte.

`utf8.offset (s, n [, i])`

Returns the position (in bytes) where the encoding of the `n`-th character of `s` (counting from position `i`) starts. A negative `n` gets characters before position `i`. The default for `i` is 1 when `n` is non-negative and size `s` + 1 otherwise, so that utf8.offset(`s`, `-n`) gets the offset of the `n`-th character from the end of the string. If the specified character is neither in the subject nor right after its end, the function returns **null**.

As a special case, when `n` is 0 the function returns the start of the encoding of the character that contains the `i`-th byte of `s`.

This function assumes that s is a valid UTF-8 string.

9.4 aconv - Internationalization

As a *plus* package, the **aconv** package is not part of the standard distribution and must be activated with the **import** statement, i.e. `import aconv`.

The package is not available for Mac OS X.

The **aconv** library allows to convert strings from one code page (character set) to another. For a list of available code pages, see **aconv.list**. It is a port to the GNU iconv package, where iconv stands for `internationalization conversion`.

Typical usage: First open a handle by passing the from code page and the to code page, in this example, we convert a text from Latin-1 to UTF-8:

```
> import aconv

> cd := aconv.open('latin1', 'utf-8');

> aconv.convert(cd, 'äöüß'):
-ä-ö-ü+í
```

After all strings have been converted, the handle must be closed.

```
> aconv.close(fd);
```

Hint for UNIX & MacOS X users: You must have the iconv package installed on your system in order to use this package.

The available functions are:

aconv.open (from, to)

Opens a new conversion descriptor, from the `from` character set (a string) to the `to` character set (also a string). Concatenating "//TRANSLIT" to the first argument will enable character transliteration and concatenating "//IGNORE" to the first argument will cause iconv to ignore any invalid characters found in the input string.

This function returns a new converter or issues an error. For a list of available character sets, see **aconv.list**. `from` and `to` may be given in upper and lower case.

`aconv.convert (cd, str)`

Converts string `str` to the desired character set. `cd` depicts the converter descriptor. This method always returns the converted string on success, and **null** and an error code otherwise:

- aconv.ERROR_NO_MEMORY
  Failed to allocate enough memory in the conversion process.
- aconv.ERROR_INVALID
  An invalid character was found in the input sequence.
- aconv.ERROR_INCOMPLETE
  An incomplete character was found in the input sequence.
- aconv.ERROR_FINALIZED
  Trying to use an already-finalized converter. This usually means that the user was tweaking the garbage collector private methods.
- aconv.ERROR_UNKNOWN
  There was an unknown error.

See also: **strings.tolatin**, **strings.toutf8**.

`aconv.close (cd [, ···])`

Closes one or more converters `cd` and for each converter successfully closed returns **true**, or **false** otherwise.

`aconv.list ()`

Returns a table of all supported codepages.

9.5 hashes - Hashes

As a *plus* package, the hashes package is not part of the standard distribution and must be activated with the **import** statement, i.e. `import hashes`.

9.5.1 Introduction

The packages computes various hashes for variable-sized strings and for numbers. All the functions require a string or number as the first argument, and - with the exception of the **hashes.md5** function - optionally the maximum number of slots in an assumed hash table as the second argument if you want the modulus of the hash value to be returned. Alternatively, you can tentatively apply **hashes.fibmod32** to the resulting hash for more evenly distributed results.

For almost each of the functions listed below an algorithm in the Agena language roughly explaining its mode of operation has been given.

9.5.2 Usefulness

With a dictionary of 517,996 surnames, where each surname consists of 7.55 characters on average, the following table shows the performance of some string hashes, computed on an Intel i-5 6500 CPU, 3.2 GHz.

Hash	Collision quotient	Collisions	Max. values per hash	Running time
adler32	1.000054	28	2	2.46
asu	1.022457	11'377	5	2.33
bkdr	1.000234	121	2	2.37
bp	3.360010	363'831	4'656	2.2
bsd	12.675476	477'130	447	2.55
cksum	1.000054	28	3	2.33
crc16	7.910631	452'515	23	3.20
crc32	1.000064	33	2	2.27
dek	1.003477	1'795	4	2.30
djb	1.000788	408	2	2.35
djb2	1.000122	63	2	2.66
djb2rot	1.000073	38	2	2.46
elf	1.022457	11'377	5	2.41
fletcher	3.447653	367'750	42	2.28
fnv	1.000070	36	2	2.24
jen	1.000058	30	2	2.39
lua	1.001793	927	3	2.22
md5	1.000000	0	1	6.94
murmur2	1.000044	23	2	2.30
murmur3	1.000075	39	2	2.30
murmur3128	1.000066	34	2	2.57
oaat	1.000098	51	2	2.17

Hash	Collision quotient	Collisions	Max. values per hash	Running time
pjw	1.022457	11'377	5	2.16
pl	1.000056	29	2	2.71
raw	1.003783	1'952	3	2.27
roaat	1.000655	339	2	2.37
rs	1.000073	38	2	2.19
sax	1.000879	455	2	2.29
sdbm	1.000064	33	2	2.31
sth	1.009873	5'064	5	2.25
strval	8.042167	453'586	8'712	2.29
varlen	1.000000	0	1	5.75

9.5.3 Summary of Operators and Functions

Numeric Hashes and Checksums

> **%, symmod, math.morton, math.modulus, math.nearmod, bytes.parity, hashes.damm, hashes.digitsum, hashes.droot, hashes.fibmod32, hashes.ftok, hashes.interweave, hashes.j32to32, hashes.jinteger, hashes.jnumber, hashes.luhn, hashes.mix, hashes.mix64, hashes.mix64to32, hashes.parity, hashes.reflect, hashes.varlen, hashes.verhoeff**

String Hashes

> **hashes.adler32, hashes.asu, hashes.bkdr, hashes.bp, hashes.bsd, hashes.cksum, hashes.crc8, hashes.crc16, hashes.crc32, hashes.dek, hashes.djb, hashes.djb2, hashes.djb2rot, hashes.elf, hashes.fletcher, hashes.fuv, hashes.jen, hashes.lua, hashes.md5, hashes.murmur2, hashes.murmur3, hashes.murmur3128, hashes.oaat, hashes.pjw, hashes.pl, hashes.raw, hashes.roaat, hashes.rs, hashes.sax, hashes.sdbm, hashes.sha512, hashes.sth, hashes.strval, hashes.sumupchars, hashes.sysv, hashes.varlen**

9.5.4 Functions

```
hashes.adler32 (s [, n [, h]])
```

Returns the Adler32 hash for string `s`. If `n` is given and non-zero, the hash is taken modulo `n` before returning. `h` by default is 65521, but may be any other non-negative integer.

`hashes.asu (s [, n [, h]])`

Returns a hash for string `s` as proposed by A. V. Aho, R. Sethi, J. D. Ullman in their book "Compilers: Principle, Techniques, and Tools", Addison-Wesley, 1988, p. 436.

If `n`, a positive integer, is given, the computed hash is taken modulo `n`. The optional argument `h` determines the initial value of the resulting hash code before the string is evaluated, and is 0 by default.

The algorithm used is equivalent to:

```
asu := proc(s :: string, n, h) is
   local g;
   n := n or 0;
   h := h or 0;
   for i in s do
      h := (h <<< 4) &+ abs i;
      g := h && 0xf0000000;
      if g <> 0 then
         h := h ^^ (g >>> 24);
         h := h ^^ g
      fi
   od;
   return if n <> 0 then h % n else h fi
end;
```

See also: **hashes.elf**.

`hashes.bkdr (s [, n [, seed]])`

Computes a hash value published by Brian Kernighan and Dennis Ritchie, for string `s`. If `n`, a positive integer, is given, the computed hash is taken modulo `n`. The optional integer `seed` determines a salt is 131 by default; you may chose other primes if necessary. The return is a number. The algorithm used is equivalent to:

```
bkdr := proc(s :: string, n, h) is
   n := n or 0;
   seed := seed or 131;  # 31, 131, 1313, 13131, 131313, etc.
   h := 0;
   for i in s do
      h := (h &* seed) &+ abs i
   od;
   return if n <> 0 then h % n else h fi
end;
```

`hashes.bp (s [, n [, h]])`

Computes a hash for string `s`; it may be useful to classify words with common endings since they have the same hash code. If `n`, a positive integer, is given, the computed hash is taken modulo `n`. The optional argument `h` determines the initial value of the resulting hash code before the string is evaluated, and is 0 by default.

The return is a number. The algorithm used is equivalent to:

```
bp := proc(s :: string, n, h) is
   n := n or 0;
   h := h or 0;
   for i in s do
      h := h <<< 7 ^^ abs i;
   od;
 return if n <> 0 then h % n else h fi
end;
```

See also: **hashes.strval**, **math.ndigits**, **math.nthdigit**.

`hashes.bsd (s [, mode])`

Returns the 8-bit or 16-bit BSD checksum for the given string `s`. The return is a non-negative integer. By default, the function computes the 16-bit checksum, a value between 0 and 65,535 if `mode` is not given or is **true**. If `mode` is set to **false**, the 8-bit checksum is computed, a value between 0 and 255.

`hashes.cksum (s [, len])`

Returns the same checksum as the UNIX cksum utility for the given string `s`. The return is a non-negative integer. By default, the full length of `s` is evaluated, but you may compute the hash for the first `len` characters by passing a second argument (an integer).

The function can be used to validate the integrity of a file but may not always detect hacker manipulation.

`hashes.collisions (s, f [, iters [, factor [, returnbag]]])`

Takes a table or sequence `s` of strings and one of the hash functions `f` and returns the mean number of collisions (a value of 1 is best), number of total slots (occupied or free), the time it took to run the procedure, and if `returnbag` is **true**, the hashing table (a bag). If `iters`, a positive integer, is not given, then the function determines the hash values only once, otherwise `iters` times. If `factor`, a positive integer or fraction, is not given, the number of slots of the virtual hash table is twice the number of elements in `s`.

The function is written in Agena (see lib/hashes.agn).

`hashes.crc8 (s [, init])`

Performs 8-bit reversed cyclic redundancy check for string `s`, starting with initial CRC value `init`, which is 0 by default. The return is a non-negative integer.

`hashes.crc16 (s [, init])`

Performs 16-bit reversed cyclic redundancy check for string `s`, starting with initial CRC value `init`, which is 0 by default. The return is a non-negative integer.

`hashes.crc32 (s [, init])`

Performs 32-bit reversed cyclic redundancy check for string `s`, starting with initial CRC value `init`, which is 0 by default. The return is a non-negative integer.

`hashes.damm (x [, true])`

If passed no option, computes the checksum of its argument `x` (an integer or string consisting of ciphers), and returns an integer in the range 0 .. 9 using the Damm algorithm. Contrary to the Luhn algorithm, it detects all single-digit errors and all adjacent transposition errors.

If passed the Boolean option **true**, the function checks whether `x` includes the correct checksum digit at its end.

If you pass an integer `x` and if $|x| >$ **math.lastcontint**, then an error will be issued, for `x` cannot be represented accurately on your system. Pass a string instead.

See also: **hashess.luhn**, **hashes.verhoeff**.

`hashes.dek (s [, n [, h]])`

Computes a hash value for string `s` proposed by Donald E. Knuth in The Art Of Computer Programming Volume 3, under the topic of sorting and search.

If `n`, a positive integer, is given, the computed hash is taken modulo `n`. The optional argument `h` determines the initial value of the resulting hash code before the string is evaluated, and is 0 by default. The algorithm used roughly resembles:

```
dek := proc(s :: string, n, h) is
   n := n or 0;
   h := h or size s;
   for i in s do
      h := ((h <<< 5) ^^ (h >>> 27)) ^^ abs i
   od;
   return if n <> 0 then h % n else h fi
end;
```

`hashes.digitsum (x [, n])`

Computes the digit sum of the integer `x` to the base `n` and returns an integer. `n` is 10 by default. If `x` is negative, the result is negative, i.e. -**hashes.digitsum**(abs(`x`), `n`) will be returned. The function is written in Agena and included in the lib/hashes.agn file.

hashes.djb (s [, n [, lsh [, h]]])

Computes the Daniel J. Bernstein hash for string s. If n, a positive integer, is given, the computed hash is taken modulo n. The optional argument h determines the initial value of the resulting hash code before the string is evaluated, and is 5381 by default. The return is a number. The algorithm used roughly resembles:

```
djb := proc(s :: string, n, sh) is
   local h;
   h :=  5381;
   n :=  n or 0;
   sh := sh or 5;
   for i in s do
      h :=(h <<< sh) &+ h &+ abs i
   od;
   return if n <> 0 then h % n else h fi
end;
```

hashes.djb2 (s [, n [, f [, h]]])

Computes a modified Daniel J. Bernstein hash for string s. If n, a positive integer, is given, the computed hash is taken modulo n. The optional argument h determines the initial value of the resulting hash code before the string is evaluated, and is 5381 by default. The return is a number. The algorithm used roughly resembles:

```
djb2 := proc(s :: string, n, f) is
   local h;
   h :=  5381;
   f :=  f or 33;
   n :=  n or 0;
   for i in s do
      h := (f &* h) ^^ abs i
   od;
   return if n <> 0 then h % n else h fi
end;
```

hashes.djb2rot (s [, n [, sh [, f [, h]]]])

Like **hashes.djb2**, but using an additional left rotation-bit shift operation; good performance, few collisions. The algorithm used is equivalent to:

```
djb2rot := proc(s :: string, n, sh, f) is
   local h;
   h :=  5381;
   f :=  f or 33;
   sh := sh or 17;
   n :=  n or 0;
   for i in s do
      h := h <<<< sh;
      h := (f &* h) ^^ abs i
   od;
   return if n <> 0 then h % n else h fi
end;
```

`hashes.droot (x [, b])`

Returns the digital root and the additive persistence for the integer x and base b. By default b is 10.

The digital root is the sum of its digits and the sum of the digits of this sum, and so forth, until the respective sum is less than b. The additive persistence is the number of summations it took to compute the root.

`hashes.elf (s [, n [, h]])`

Similar to **`hashes.asu`**, but optimised for 32-bit CPUs, commonly used in UNIX systems. The code is equivalent to:

```
elf := proc(s :: string, n, h) is
   local x;
   n := n or 0;
   h := h or 0;
   for i in s do
      h :=(h <<< 4) &+ abs i;
      x := h && 0xf0000000;
      if x <> 0 then
         h := h ^^ (x >>> 24);
      fi;
      h := h && ~~(x)
   od;
   return if n <> 0 then h % n else h fi
end;
```

`hashes.fibmod32 (k, m [, true])`

Returns an unsigend 32-bit integer hash value for the non-negative integer k and the given number of slots m - also a non-negative integer - that *may* be more evenly distributed than just computing k % m, but not necessarily. The function uses Fibonacci hashing, and returns a value that is equivalent to:

```
fibmod := proc(k :: nonnegint, m :: nonnegint) is
   local p := k *(math.Phi – 1);
   return bytes.numto32(m * frac(p))
end;
```

The result is in the range 0 .. m - 1 if m is odd; if m is even, the result is always in the range 1 .. m - 1, unless k = 0 or m = 0 where the function returns 0.

Note that with a < b, fibmod32(a, m) is not necessarily less than fibmod32(b, m).

If you pass the optional third argument **true**, then the results are always the same across different platforms - due to performance reasons, the default is **false**.

See also: **%**, **math.modulus**.

hashes.fletcher (s, [mode [, len]])

If `mode` is not given or is **true**, returns the position-dependent 16-bit checksum of a string s according to Fletcher's algorithm using an internal 32-bit accumulator, and returns an integer. The 360th and all succeeding characters are ignored.

If `mode` is **false**, returns the position-dependent 8-bit checksum using an internal 16-bit accumulator, and returns an integer in the range [257, 65535]. The 21st and all succeeding characters are ignored.

If the option `len` is given, only the first `len` characters are processed.

hashes.fnv (s [, n])

Computes the Fowler-Noll-Vo hash for string `s`. If `n`, a positive integer, is given, the computed hash is taken modulo `n`. The return is a number. The algorithm used is equivalent to:

```
fnv := proc(s :: string, n) is
   local h;
   h := 2166136261;
   n := n or 0;
   for i in s do
      h := (h &* 16777619) ^^ abs i
   od;
   return if n <> 0 then h % n else h fi
end;
```


hashes.ftok (inode, device [, id, [, n]])

Computes the System V IPC (Inter Process Communications) key. `inode`, `device` and optional `id` are all 4-byte signed integers, with `id` defaulting to 0. The return is an integer equivalent to, in signed bits mode:

$$(\texttt{inode} \mathbin{\&\&} \texttt{0xffff}) \mathbin{||} ((\texttt{device} \mathbin{\&\&} \texttt{0xff}) <<< 16) \mathbin{||} ((\texttt{id} \mathbin{\&\&} \texttt{0xffu}) <<< 24)$$

If `n` is given and non-zero, the hash is taken modulo `n` before returning.

See also: **os.ftok**.

hashes.interweave (x [, option [, mask [, sh [, n]]]])

Splits a number `x` into its higher and lower unsigned 4-byte words hx and lx and applies one of the following binary operations on them: `'or'` (the default), `'and'`, `'xor'`.

By passing a non-negative `mask` as the optional third argument, the mask is applied to the intermediate result, the default is 0xFFFFFFFF.

If a fourth positive `sh` integer is given, the intermediate result is right-shifted `sh` bits; if `sh` is a negative integer, it is left-shifted `sh` bits. If `sh` is 0 (the default), there is no shift.

If a fifth argument `n` is given, a positive integer, the intermediate result is taken modulus `n`. The default is 1.

Thus, with hx, lx := bytes.numwords(x):

- `option = 'or'`: (((hx || lx) && `mask`) >>> `sh`)   % n, if `sh` > 0,
- `option = 'and'`: (((hx && lx) && `mask`) >>> `sh`)   % n, if `sh` > 0,
- `option = 'xor'`: (((hx ^ ^ lx) && `mask`) >>> `sh`)   % n, if `sh` > 0,

and

- `option = 'or'`: (((hx || lx) && `mask`) <<< |`sh`|)   % n, if `sh` < 0,
- `option = 'and'`: (((hx && lx) && `mask`) <<< |`sh`|)   % n, if `sh` < 0,
- `option = 'xor'`: (((hx ^ ^ lx) && `mask`) <<< |`sh`|) % n, if `sh` < 0.

hashes.j32to32 (x [, n])

Hashes an unsigned 4-byte integer `x` (i.e. in the range 0 .. 2^{32} - 1) to yet another integer in the same range, Julia-style.

If a second argument `n` is given, a positive integer, the intermediate result is taken modulus `n`. The default for `n` is 1.

See also: **hashes.jinteger**.

hashes.jen (s [, n])

Computes the Bob Jenkins' hash (96 bit Mix Function) for string `s`. If `n`, a positive integer, is given, the computed hash is taken modulo `n`. The return is a number. Please see the C hashes.c source file for its implementation.

hashes.jinteger (x [, h])

Value-based hashing of an unsigned 4-byte integer `x`, with seed `h` which by default is 4,294,967,295 = 2^{32} - 1, ported from the Julia language.

See also: **hashes.jnumber**, **hashes.j32to32**.

hashes.jnumber (x [, n [, option]])

Maps a number `x` to one or two unsigned 4 byte integers, Julia-style. If `n`, a positive integer, is given, the computed hashes are taken modulo `n`. By default, only one unsigned 4-byte integer will be returned. If you pass **true** for `option` then the function will split `x` into its higher and lower unsigned 4-byte words and returns

unsigned 4-byte integer hashes for each of them. In this case, the second return is equal to the result of **hashes.jnumber** when called without this `option`.

See also: **bytes.numwords**, **hashes.jinteger**.

`hashes.lua (s [, n])`

Returns the hash, an integer, Lua/Agena internally computes for string `s`. If `n`, a positive integer, is given, the computed hash is taken modulo `n`. It is an adaption of the Shift-Add-XOR hash. This variant chooses the length of the string as its seed, not a fixed value, and scans from the right to the left. See also: **hashes.sax**.

`hashes.luhn (x [, true])`

If passed no option, computes the checksum of its argument `x` (an integer or string consisting of ciphers), and returns an integer in the range 0 .. 9 using the Luhn formula, which is used to validate credit card numbers, IMEIs or some social security numbers.

If passed the Boolean option **true**, the function checks whether `x` includes the correct checksum digit at its end.

If you pass an integer `x` and if $|x| >$ **math.lastcontint**, then an error will be issued, for `x` cannot be represented accurately on your system. Pass a string instead.

The Luhn formula does not recognise the transposition 09 vs. 90, nor does it detect twin 22 vs. 55, 33 vs. 66, and 44 vs. 77.

See also: **hashess.damm**, **hashes.verhoeff**.

`hashes.md5 (s [, anyoption])`

Computes the MD5 hash for strings `s`. The return is a string of 32 characters that represent 16 pairs of hexagesimal numbers where the alphabetical letter is in upper-case.

If any option is given, the MD5 hash for file `s` will be returned.

See also: **hashes.varlen**.

`hashes.mix (a, b, c)`

The function mixes three non-negative integers `a`, `b`, `c` assumed to be 32-bit and returns an integer.

`hashes.mix64 (x [, n])`

Computes the 64-bit mix for number `x`. If `n`, a positive integer, is given, the computed hash is taken modulo `n`. The return is a number. See also: **hashes.mix64to32**.

`hashes.mix64to32 (x [, n])`

Computes the 64-bit mix for number `x`. If `n`, a positive integer, is given, the computed hash is taken modulo `n`. The return is a number. See also: **hashes.mix64**.

`hashes.murmur2 (s [, n])`

Returns MurmurHash2 for string `s`. If `n`, a positive integer, is given, the computed hash is taken modulo `n`. Note that the function returns different values on little-endian and Big Endian machines.

See also: **hashes.murmur3**, **hashes.murmur3128**.

`hashes.murmur3 (s [, n])`

Computes MurmurHash3 using 32-bit unsigned integers internally, for the given string `s` and returns an integer. If `n`, a positive integer, is given, the computed hash is taken modulo `n`. Note that the function returns different values on little-endian and big-endian machines.

See also: **hashes.murmur2**, **hashes.murmur3128**.

`hashes.murmur3128 (s [, n [, seed]])`

Computes MurmurHash3 using 128-bit unsigned integers internally, for the given string `s` and returns four unsigned 32-bit integers. If `n`, a positive integer, is given, the computed hash is taken modulo `n`. `seed`, if not given, is 0x9747b28c by default. Note that the function returns different values on little-endian and big-endian machines. (In OS/2 and Raspberry Pi the function always returns an error.)

See also: **hashes.murmur2**, **hashes.murmur3128**.

`hashes.oaat (s [, n [, h]])`

Computes the One-at-a-Time hash for string `s`. If given, `n` must be a positive integer. The optional argument `h` determines the initial value of the resulting hash code before the string is evaluated, and is 0 by default. The return, which may vary across platforms, is a number. The algorithm used is equivalent to:

```
hashmask := << n -> (1 <<< n) - 1 >>
```

```
oaat := proc(s :: string, n) is
   local h := 0;
   n := n or 0;
   for i in s do
      inc h, abs i;
      inc h, h <<< 10;
      h := h ^^ (h >>> 6)
   od;
   inc h, h <<< 3;
   h := h ^^ (h >>> 11);
   inc h, h <<< 15;
   return if n <> 0 then h && hashmask(n) else h fi
end;
```

See also: **hashes.roaat**.

hashes.parity (x)

Returns a byte with even parity for the non-negative integer x, and returns an integer in the range [0, 255]. See also: **bytes.parity32**.

hashes.pjw (s [, n [, h]])

Computes the P. J. Weinberger Hash for string s. If n, a positive integer, is given, the computed hash is taken modulo n. The optional argument h determines the initial value of the resulting hash code before the string is evaluated, and is 0 by default.

The return is a number.

hashes.pl (s [, n [, f [, h]]])

Computes Paul Larson's hash of Microsoft Research for string s. If n, a positive integer, is given, the computed hash is taken modulo n. The optional argument h determines the initial value of the resulting hash code before the string is evaluated, and is 0 by default. The return is a number. The algorithm is equivalent to:

```
pl := proc(s :: string, n, f, h) is
   local h := 0;
   f := f or 101;
   n := n or 0;
   h := h or 0;
   for i in s do
      h :=(h &* f) &+ abs i
   od;
   return if n <> 0 then h % n else h fi
end;
```

With initial $h=5381$ and $f=33$, emulates the GNU hash.

hashes.raw (s [, n [, h]])

Computes a self-invented hash for string s. If n, a positive integer, is given, the computed hash is taken modulo n. The optional argument h determines the initial

value of the resulting hash code before the string is evaluated, and is 0 by default. The return is a number. The algorithm used is equivalent to:

```
raw := proc(s :: string, n, h) is
   n := n or 0;
   h := h or 0;
   for i in s do
      h := 38 &* (h <<< 1) &+ abs i &- 63
   od;

   return if n <> 0 then h % n else h fi
end;
```

hashes.reflect (x [, n])

Reorders the bits of the n-bit integer x by reflecting them about the middle position. By default, n is 32, but may be any other integer in [1, 32]. The return is an integer.

hashes.roaat (s [, n [, h]])

Like **hashes.oaat**, but uses bit rotation internally instead of simple bit shifts. The result may vary across platforms.

hashes.rs (s [, n [, h]])

Computes a hash for string s. If n, a positive integer, is given, the computed hash is taken modulo n. The optional argument h determines the initial value of the resulting hash code before the string is evaluated, and is 0 by default. The return is a number. The algorithm used is equivalent to:

```
rs := proc(s :: string, n, h) is
   local a, b := 63689, 378551;
   n := n or 0;
   h := h or 0;
   for i in s do
      h := h &* a &+ abs i;
      a := a &* b
   od;
   return if n <> 0 then h % n else h fi
end;
```

hashes.sax (s [, n [, h]])

Computes the Shift-Add-XOR hash for string s. If n, a positive integer, is given, the computed hash is taken modulo n. The optional argument h determines the initial value of the resulting hash code before the string is evaluated, and is 5381 by default. The return is a number. The algorithm used is equivalent to:

```
sax := proc(s :: string, n) is
   local h := 5381;
   n := n or 0;
   for i in s do
      h := h ^^ ((h <<< 5) + (h >>> 2) + abs i)
   od;
   return if n <> 0 then h % n else h fi
end;
```

hashes.sdbm (s [, n [, h]])

Computes the ndbm database library hash for string `s`. If `n`, a positive integer, is given, the computed hash is taken modulo `n`. The optional argument `h` determines the initial value of the resulting hash code before the string is evaluated, and is 0 by default. The return is a number. The algorithm uses a public-domain implementation. The algorithm used is equivalent to:

```
sdbm := proc(s :: string, n) is
   local h := 0;
   n := n or 0;
   for i in s do
      h := abs i &+ (h <<< 6) &+ (h <<< 16) &- h
   od;
   return if n <> 0 then h % n else h fi
end;
```

hashes.sha512 (s [, salt [, rounds]])

Calculates a SHA512 cryptographic hash for string `s`, optionally using a `salt` of type string, and the optional number of `rounds` to be taken. `salt` by default is the empty string and `rounds` is 5000.

The first return is the hash itself, and the second return includes the control parameters `salt` and `rounds` plus the first result. Thus, the second return is the same as the output of the mkpasswd UNIX command.

In case of errors, the function returns **fail**.

The function is very slow.

hashes.sth (s [, n [, h]])

Computes the s-th hash for string `s`. If `n`, a positive integer, is given, the computed hash is taken modulo `n`. The optional argument `h` determines the initial value of the resulting hash code before the string is evaluated, and is 0 by default. The return is a number. The algorithm has been published at StackOverflow. The algorithm is equivalent to:

```
sth := proc(s :: string, n) is
   local h := 0;
   n := n or 0;
   for i in s do
      h := (h <<< 6) ^^ (h >>> 26) ^^ abs i
   od;
   return if n <> 0 then h % n else h fi
end
```

hashes.strval (s [, sh [, h]])

Computes a hash with many collisions, useful to classify words with common endings since they have the same hash code. `s` denotes the string to be hashed, `sh` the left-shift, which is -8 by default; and `h` the initial hash value before computation starts, 0 by default. The algorithm used is equivalent to:

```
strval := proc(s :: string, sh, h) is
   sh := n or 8;
   h := h or 0;
   for i in s do
      h := h <<< sh;
      h := h &+ abs i
   od;
   return h
end;
```

See also: **hashes.bp**, **math.ndigits**, **math.nthdigit**.

hashes.sumupchars (s [, f [, g]])

Sums up all the ASCII values in string `s` and returns the result as a positive integer. The sum is expressed as an unsigned 32-bit integer, so keep overflows in mind.

Instead of just adding the plain ASCII values, you might optionally apply function `f` to the first character in `s`, and function `g` to the second character in `s`, then `f` again on the third character, `g` on the fourth character, and so forth. Example:

```
> import bytes;

> sum := hashes.sumupchars("agena",
>    << x ->(x && 0xff) <<< 8 >>,
>    << x -> x && 0xff >>):

> ~~(sum + (sum >>> 16)):
```

Note that the Internet checksum, even if strictly implemented according to RFC 1071, has by far more collisions than the other hash functions available in this package.

hashes.sysv (s [, n [, f [, h]]])

For string `s`, computes the System V hash to access libraries via dynamic symbol tables on UNIX. If `n`, a positive integer, is given, the computed hash is taken modulo `n`. The optional positive integer f is the factor to multiply intermediate results, see algorithm below, and is 16 by default. The optional argument seed `h` determines the salt and is 0 by default; you may chose other primes if necessary.

The return is a number.

The algorithm used is equivalent to:

```
sysv := proc(s :: string, n, f, h) is
   n := n or 0;
   f := f or 16;
   h := h or 0;
   for i in s do
      h := f &* h &+ abs i;
      h := h ^^ ((h >>> 24) && 0xf0)
   od;
   h := h && 0xfffffff;
   return if n <> 0 then h % n else h fi
end;
```

For GNU hash, see **hashes.pl**.

hashes.varlen (x, salt [, n])

Computes a variable-length integer hash for string or number x and string salt. If the optional positive integer n is given, the computed hash is taken modulo n. Depending on the given keyword, the number of collisions might be zero, so this function is an alternative to **hashes.md5**.

hashes.verhoeff (x [, true])

If passed no option, computes the checksum of its argument x (an integer or string consisting of ciphers), and returns an integer in the range 0 .. 9 using the Verhoeff algorithm. Contrary to the Luhn algorithm, it detects all single-digit errors, and all accidental transposition involving two adjacent ciphers.

If passed the Boolean option **true**, the function checks whether x includes the correct checksum digit at its end.

If you pass an integer x and if $|x| >$ **math.lastcontint**, then an error will be issued, for x cannot be represented accurately on your system. Pass a string instead. The function also returns an error, if a non-digit is included in string x.

See also: **hashes.damm**, **hashes.luhn**.

9.6 bloom - Bloom Filter

As a *plus* package, the **bloom** package is not part of the standard distribution and must be activated with the **import** statement, i.e. `import bloom`.

9.6.1 Introduction

This package implements the Bloom filter, a dictionary containing bit signatures of its individual strings (words).

A Bloom filter is a memory-efficient mean to check whether a string *probably* is part of a dictionary or whether it is *definitely not* part of the dictionary, with acceptable query times. It consumes less memory than the original dictionary of strings and can be used to prevent unnecessary access to the file system on which the actual dictionary resides, for example in dBASE III+, binary or text files.

With respect to this package, a dictionary does not depict an Agena table dictionary, but just a list of strings, e.g.: "Akatsuki", "Chandrayaan", "Chang'e", "Mars Express", "Venera", "Voyager".

Depending on the size of the Bloom filter, the hash string function used, and the number of internal iterations - i.e. number of `salts` - when inserting or reading values, around 80 % of memory can be saved with only around 5 % of the words to be actually looked up in the original dictionary. Bloom filter lookup takes around a third more running time than searching Agena built-in data structures.

Technically, the hash value of a string - see **hashes** package for a variety of string hash functions - is converted into a bit signature that is stored to slots in the Bloom filter. Internally, the Bloom filter implemented here uses four unsigned bytes for each slot (C type uint32_t). The string hash function should produce the least number of collisions.

You cannot delete values from a Bloom filter. Also, you cannot change the number of slots of the bloom filter or the number of salts.

You may use the package as follows:

1. Determine the number of entries `s` in your original dictionary `d`.

2. Create a Bloom filter `b` with `s`\4 slots and 4 salts:

```
b := bloom(s \ 4, 4);
```

3. Insert all entries `str` of your dictionary into Bloom filter `b` using a string hash function, e.g.:

```
for str in d do
   bloom.include(b, hashes.sdbm(str))
od;
```

4. Query the Bloom filter for any entry, using the same hash function:

```
result := hashes.find(b, hashes.sdbm('Zond'));

if result = false then
   print('entry really not included')
else
   print('entry probably included, search original dictionary.')
fi;
```

5. Query a Bloom filter slot, with an index counting from 1:

```
b[1]:
```

of just output all slots with **bloom.toseq**.

6. Check the state of the bloom filter `b`:

```
bloom.attrib(b):
```

9.6.2 Functions

`bloom.attrib (b)`

Returns various information on the Bloom filter:

- key `'size'`: number of internal slots of the bloom filter, the first argument to **bloom.new**.
- key `'salts'`: number of internal hash functions (salts) applied to a word when computing the signature, the second argument to **bloom.new**.
- key `'wordsincluded'`: number of words included into the filter. If the signature of a word is already included, it is not counted.
- key `'collisions'`: number of collisions detected when trying to include a word into the filter, for its internal signature is already present. If a word has already been included in the filter, its collision is being counted nevertheless.
- key `'bytes'`: size of the whole Bloom filter userdata in bytes.

`bloom.get (b, i)`

With a bloom filter `b`, returns the value stored at `b[i]`, where `i`, the index, is an integer counting from 1.

See also: **bloom.toseq**.

bloom.find (b, hash)

Checks whether a string converted to the `hash` value is part of a dictionary of strings represented by Bloom filter userdata `b`. The function returns **true** or **false**, where **false** means that the string is *definitely not included* in the original dictionary, and **true** means it is *probably part of* the original dictionary.

Example: `bloom.find(b, hashes.pl('Soyuz'))`.

See also: **bloom.include**.

bloom.include (b, hash)

Inserts the `hash` value (an integer) of a string into the Bloom filter `b`, a userdata. By default, the function returns nothing.

If a hash value has already been inserted, nothing happens.

If the optional third argument is **true**, internal information will be returned: the last internal subhash - an integer - computed before inserting the signature of the string into the Bloom filter, and a table with the keys representing the slot indices of the Bloom filter modified (an integer starting from 1) and the respective bit position set to 1 (counting from 0, from the right of the bit field).

Example: `bloom.include(b, hashes.pl('Soyuz'))`.

See also: **bloom.find**.

bloom.new (n, salts)

Creates a Bloom filter, of type userdata, consisting of `n` slots. The number of salts internally applied when inserting or searching the hash value of a string is given by `salts`, a positive integer in the range [1, 65]. If `salts` is 1, then no salt is applied, otherwise (`salts` - 1) salts are applied.

With a large list of surnames, for example, `n` should be at least a fourth of the number of words contained in the dictionary, and `salts` should be 4.

See also: **bloom.attrib**, **bloom.toseq**.

bloom.toseq (b)

Receives a Bloom filter `b` and converts its internal slots into a sequence of integers, the return.

See also: **bloom.get**.

Chapter **Ten**

Structures

10 Structures

10.1 Tables

Summary of Functions:

Queries

> **countitems, empty, filled, in, notin, size, tables.getsize, tables.maxn, type, typeof**.

Retrieving Values

> **getentry, unique, unpack, values, tables.array, tables.borders, tables.entries, tables.hash, tables.indices, tables.parts**.

Operations

> **copy, copyadd, map, move, purge, put, qsumup, remove, sumup, select, selectremove, shift, sort, sorted, subs, swap, zip, tables.include, tables.move, tables.reshuffle**.

Relational Operators

> **=, ==, ~=, <>, ~<>**.

Cantor Operations

> **intersect, minus, subset, union, xsubset**.

Miscellaneous

> **tables.dimension, tables.allocate, tables.newtable**.

10.1.1 Operators and Functions

Most of the following functions have been built into the kernel as unary operators, with the exception of **map** and **zip**.

```
copy (t)
```
The function copies the entire contents of a table `t` into a new table. See Chapter 8 for more information.

copyadd (t [, ···])

See Chapter 8.

countitems (item, t)

countitems (f, t [, ···])

In the first form, counts the number of occurrences of an `item` in the table `t`.

In the second form, by passing a function `f` with a Boolean relation as the first argument, all elements in the structure `t` that satisfy the given relation are counted. If the function has more than one argument, then all arguments *except the first* are passed right after the name of table `t`.

The return is a number. The function may invoke metamethods.

See also: **select**.

empty (t)

Checks whether table `t` does not contain any element. The return is **true** or **false**. The operator works with dictionaries, as well. See also: **filled**.

filled (t)

Checks whether table `t` contains at least one element. The return is **true** or **false**. The operator works with dictionaries, as well. See also: **empty**.

getentry (t [, k_1, ···, k_n])

Returns the entry $t[k_1, ···, k_n]$ from the table `t` without issuing an error if one of the given indices k_i (second to last argument) does not exist. See also **rawget**.

join (t [, sep [, i [, j]]])

Concatenates all string values in the table `t` in sequential order and returns a string: `t[i]` & `sep` & `t[i+1]` ··· & `sep` & `t[j]`. The default value for `sep` is the empty string, the default for `i` is 1, and the default for `j` is the length of the table. The function issues an error if `t` contains non-strings.

Use the **tostring** function if you want to concatenate other values than strings, e.g.:

```
> join(map(tostring, [1, 2, 3])):
123
```

`map (f, t [, ···] [, true])`

Maps the function `f` on all elements of a table `t`. See **map** in Chapter 8 for more information.

See also: **countitems**, **remove**, **select**, **selectremove**, **subs**, and **zip**.

`move (t1, start, stop, newidx [, t2])`

Copies elements from table `t1` to table `t2`, performing the equivalent to the following multiple assignment: `t2[newidx],··· = t1[start], ···, t1[stop]`. The default for `t2` is `t1`, i.e. elements are shifted in the same table. The destination range can overlap with the source range.

Returns the destination table `t2`.

See also: **purge**, **put**.

`purge (t [, pos])`

`purge (t, a, b)`

Removes from table `t` the element at position `pos`, shifting down other elements to close the space, if necessary. Returns the value of the removed element. The default value for `pos` is n, where n is the length of the table, so that a call `purge(t)` removes the last element of `t`.

In the second form, removes all elements starting from index `a` to index `b` (inclusive), moving excess elements down to close the space; the function automatically performs a garbage collection after shifting. In the 2nd form, nothing will be returned.

Use the **delete** *element* **from** *table* statement if you want to remove any occurrence of the table value *element* from a table.

Note that the function only works if the table is an array, i.e. if it has positive integral and consecutive keys only.

See also: **move**, **put**, **shift**, **swap**.

`put (t, [pos,] value)`

Inserts element `value` at position `pos` in table `t`, shifting up other elements to open space, if necessary. The default value for `pos` is n+1, where n is the current table size, so that a call `put(t, value)` inserts `value` at the end of `t`.

Use the **insert** *element* **into** *structure* statement if you want to add an element at the current end of a table, for it is much faster.

The function returns the modified structure.

See also: **move**, **prepend**, **purge**.

`qsumup (t)`

Raises all numeric values in table `t` to the power of 2 and sums up these powers. See **qsumup** in Chapter 8 for more information. See also: **sumup**.

`remove (f, t [, ··· [, newarray=true] [, inplace=true]])`

Returns all values in table `t` that do not satisfy a condition determined by function `f`. See **remove** in Chapter 8 for more information. See also: **map**, **select**, **selectremove**, **subs**, **zip**.

`sumup (t)`

Sums up all numeric values in table `t`. See **sumup** in Chapter 8 for more information. See also: **qsumup**.

`select (f, t [, ··· [, newarray=true] [, inplace=true]])`

Returns all values in table `t` that satisfy a condition determined by function `f`. See **select** in Chapter 8 for more information. See also: **map**, **remove**, **selectremove**, **subs**, **zip**.

`selectremove (f, t [, ··· [, newarray=true]])`

Returns all values in table `t` that satisfy and do not satisfy a condition determined by function `f`, in two tables. See **selectremove** in Chapter 8 for more information.

See also: **map**, **remove**, **select**, **subs**, **zip**.

`size (t)`

Returns the number of actual entries in the array and hash parts of table `t`. The operator returns a number and conducts a linear traversal.

See also: **environ.attrib**, **tables.getsize**.

`shift (t, a, b)`

Moves an element in the table array `t` from position `old` to `new`, with `old`, `new` integers, shifting all the other elements accordingly - which might also cause a rotation. The function returns nothing.

See also: **move**, **purge**, **swap**.

`sort (t [, comp])`

Sorts table `t` in a given order, and in-place. See **sort** in Chapter 8 for more information.

See also: **sorted**, **skycrane.sorted**, **stats.issorted**, **stats.sorted**.

`sorted (t [, comp])`

Sorts table elements in `t` in a given order, but - unlike sort - not in-place, and non-destructively. See **sorted** in Chapter 8 for more information.

See also: **sort**, **skycrane.sorted**, **stats.issorted**, **stats.sorted**.

`subs (x:v [, ···], t [, true])`

Substitutes all occurrences of value `x` in table `t` with value `v`. See **subs** in Chapter 8 for more information.

See also: **map**, **remove**, **select**, **zip**.

`swap (t, a, b)`

Swaps the table array `t` entries at index positions `a` and `b`, with `a`, `b` integers. The function returns nothing.

See also: **move**, **purge**.

`unique (t)`

The function removes all holes (`` `missing keys` ``) in the array part of table `t` and removes multiple occurrences of the same value, if present. See **unique** in Chapter 8 for more information.

`values (t, i₁ [, i₂, ···]])`

Returns the elements from the given table `t` in a new table. This function is equivalent to

```
return [ i₁ ~ t[i₁], i₂ ~ t[i₂], ··· ]
```

See also: **ops**, **select**, **unpack**.

```
zip (f, t1, t2)
```

This function zips together two tables `t1`, `t2` by applying the function `f` to each of its respective elements. See Chapter 8 for more information.

See also: **map**, **remove**, **select**, **subs**.

The following functions have been built into the kernel as binary operators.

Please note that the operators returning a Boolean work in the Cantor way, i.e. `{1, 1} = {1}` → true, `{1, 2} xsubset {1, 1, 2, 2, 3, 3}` → true.


```
t1 = t2
```

This equality check of two tables `t1`, `t2` first tests whether `t1` and `t2` point to the same table reference in memory. If so, it returns **true** and quits.

If not, the operator then checks whether `t1` and `t2` contain the same values without regard to their keys, and returns **true** or **false**. In this case, the search is quadratic.

See also: **environ.isequal**.


```
t1 == t2
```

This strict equality check of two tables `t1`, `t2` first tests whether `t1` and `t2` point to the same table reference in memory. If so, it returns **true** and quits.

If not, the operator then checks whether `t1` and `t2` contain the same number of elements and whether all key~value pairs in the tables are the same. In this case, the search is linear.

See also: **environ.isequal**.


```
t1 ~= t2
```

This approximate equality check of two tables `t1`, `t2` first tests whether `t1` and `t2` point to the same table reference in memory. If so, it returns **true** and quits.

If not, the operator then checks whether `t1` and `t2` contain the same number of elements and whether all key~value pairs in the tables are approximately equal (please see **approx** for further details). In this case, the search is linear.


```
t1 <> t2
```

This inequality check of two tables `t1`, `t2` first tests whether `t1` and `t2` do not point to the same table reference in memory. If so, it returns **true** and quits.

If not, the operator then checks whether `t1` and `t2` do not contain the same values, and returns **true** or **false**. In this case, the search is quadratic.

`t1 ~<> t2`

Approximate inequality check, the negation of the ~= operator.

`c in t`

Checks whether the table `t` contains the value `c` and returns **true** or **false**. The search is linear.

See also: **notin** operator, **binsearch** for binary search.

`c notin t`

Checks whether table `t` does not contain the value `c` and returns **true** or **false**. The search is linear.

See also: **in** operator.

`t1 intersect t2`

Searches all values in `t1` that are also values in `t2` and returns them in a new table. The search is quadratic, so you may use **bintersect** instead if you want to compare large tables since **bintersect** performs a binary search. The key ~ value pairs in the *hash* part of a table are treated as being unique. If `t1` has a metatable and/or a user-defined type, then they will be copied to the result; otherwise the function will try to copy them from `t2`.

`t1 minus t2`

Searches all values in table `t1` that are not values in table `t2` and returns them as a new table. The search is quadratic, so you may use **bminus** instead if you want to compare large tables since **bminus** performs a binary search. The key ~ value pairs in the *hash* part of a table are treated as being unique. If `t1` has a metatable and/or a user-defined type, then they will be copied to the result; otherwise the function will try to copy them from `t2`.

`t1 subset t2`

Checks whether all values in table `t1` are included in table `t2` and returns **true** or **false**. The operator also returns **true** if `t1 = t2`. The search is quadratic.

`t1 union t2`

Concatenates two tables `t1` and `t2` simply by copying all its elements - even if they occur multiple times - to a new table. The key ~ value pairs in the *hash* part of a table are treated as being unique. If `t1` has a metatable and/or a user-defined

type, then they will be copied to the result; otherwise the function will try to copy them from `t2`.

`t1 xsubset t2`

Checks whether all values in table `t1` are included in table `t2` and whether `t2` contains at least one further element, so that the result is always **false** if `t1 = t2`. The search is quadratic.

See also: **bintersect**, **bisequal**, **bminus**, **purge**, **put** in Chapter 8 Basic Functions.

`f @ t`

The operator maps a function `f` to all the values in table `t` and returns a table as the result. `f` must be a univariate function and return only one value. If `t` has metamethods or user-defined types, the return will also have them.

Examples:

```
> << x -> x^2 >> @ [1, 2, 3]:
[1, 4, 9]

> << x -> x > 0 >> @ [1, 2, 3]:
[true, true, true]
```

See also: **$** and **$$** operators.

`f $ t`

Returns all values in table `t` that satisfy a condition determined by function `f`. `f` should be a univariate function and return at least one value. In the multivariate case, all results but the first are ignored. The return might include holes.

```
> << x -> x > 1 >> $ [1, 2, 3]:
[2, 3]
```

If present, the function also copies the metatable and user-defined type of `t` to the new table.

See also: @ operator, **countitems**, **descend**, **map**, **remove**, **selectremove, subs**, **unique**, **values**, **zip.**

`f $$ t`

Checks whether at least one element in table `t` satisfies the condition defined by function `f` and returns **true** or **false**. `f` should be a univariate function and return at least one value. In the multivariate case, all results but the first are ignored.

```
> << x -> x < 1 >> $$ [1, 2, 3]:
false
```

The return might include holes.

See also: @ operator, **countitems**, **descend**, **map**, **remove**, **selectremove, subs**, **unique**, **values**, **zip.**

10.1.2 tables Library

This library provides generic functions for table manipulation. It provides all its functions inside the table `tables`.

Most functions in the table library assume that the table represents an array or a list. For these functions, when we talk about the 'length' of a table we mean the result of the length operator.

tables.allocate (t, key$_1$, value$_1$ [, key$_2$, value$_2$, $\cdots$, key$_n$, value$_n$])

Sets the specified keys and values to table `t`, i.e. `t`$_{[key_k]}$ `:= value`$_k$. Note that if a key is given multiple times, then only the first occurrence of the key in the argument sequence is processed. The function returns nothing.

See also: **tables.include**.

tables.array (t)

Returns the array part of table `t` in a new table, with all key~value pairs preserved.

See also: **tables.hash, tables.parts**.

tables.borders (t [, option])

By default, returns the smallest and largest assigned integral index - in this order - in the array part of a table `t`.

If any `option` is given, then the function determines the smallest and largest assigned integral index in both the array and hash part of table `t`. Note that this is slower since the entire hash part has to be searched linearly.

If zeros are returned, the array or the array and hash part of the table is empty.

See also: **environ.attrib, tables.getsize, tables.indices, tables.maxn**.

```
tables.dimension (a:b [, c:d, ···] [, default])
tables.dimension (a:b [, c:d, ···] [, init = default])
```

In the first form, creates a table of any dimension with arbitrary index ranges `a:b` etc. with `a`, `b`, etc. integers, and an optional `default` for all its entries. `default` must not be a pair.

In the second form the initialiser may be given as the option "init = `default`", which allows to also use pairs as a default.

If the initialiser is a structure, i.e. table, set, sequence or register, then individual copies of the initialiser are created to avoid referencing to the same structure.

See also: **tables.newtable**, **create table**/**dict** statements.

```
tables.entries (t)
```

Returns all entries of table `t` (not its keys) in a new table array. Its second result, a Boolean, indicates whether a value has been found in the hash part of `t`.

See also: **tables.indices**, **unique**, **whereis**.

```
tables.getsize (t [, option])
```

Returns a guess on the number of elements in a table `t`. If any `option` is given, the function additionally returns a Boolean indicator on whether a table contains an allocated hash part, and a Boolean indicator on whether **null** has been assigned to a table. The latter return is not foolproof, especially if a table value has been deleted with a raw assignment, e.g. t[2] := null;

The function is useful to determine the size of a table much more quickly than the **size** operator does, using a logarithmic instead of linear method, but may return incorrect results if the array part of a table has holes. It also does not count the number of elements in the hash part of a table.

See also: **size**, **tables.getsizes**.

```
tables.getsizes (t [, option])
```

If any `option` is given, returns the actual number of elements currently stored in the array and hash part. If no `option` is given, then an estimate of the number of elements in the array part will be returned, and 0 for the hash part as this cannot be estimated.

Returns two integers: the first for the array part, the second for the hash part.

See also: **size**, **tables.getsize**.

`tables.hash (t)`

Returns the hash part of table `t` in a new table with all key~value pairs preserved.

See also: **tables.array**, **tables.parts**.

`tables.include (t, key, value [, ···])`

Inserts values into a subtable of table `t`. If `t[key]` already represents a table, `value` is added to the end of its array part. If `t[key]` is unassigned, then it creates a new subtable and inserts `value` into it, which is equivalent to the pseudo code:

```
for i from 3 to nargs do
   if assigned t[key] then
      insert <argument_i> into t[key]
   else
       t[key] := [<argument_i>]
   end
fi
```

The function returns nothing.

See also: **copyadd**, **bags.include**, **tables.allocate**.

`tables.indices (t [, option])`

Returns all keys of table `t` in an unsorted new table.

If you pass any optional argument, the function will return the integral indices of a table only. In this case, the second result, a Boolean, indicates whether at least one integral key has been found in the hash part, so you might sort the table if needed. This mode is 40 % faster than the standard mode of the function.

See also: **tables.borders**, **tables.entries**, **whereis**.

`tables.maxn (t)`

Returns the largest positive numerical index of the given table `t`, or zero if the table has no positive numerical indices. (To do its job this function does a linear traversal of the whole table.) See also **tables.borders**, which is faster with arrays.

`tables.move (t1, start, stop, newidx [, t2])`

Copies elements from the table `t1` to the table `t2`, performing the equivalent to the following multiple assignment: `t2[newidx]`,··· = `t1[start]`, ···, `t1[stop]`. The default for `t2` is `t1`, i.e. elements are shifted in the same table. The destination range can overlap with the source range.

Returns the destination table `t2`.

Example: The following statement copies four elements in table a from position 3 up to and including 6 to a new table b, starting with index 1:

```
> a := ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'];

> b := tables.move(a, 3, 6, 1, []);

> b:
[c, d, e, f]
```

The next statement copies four elements in a to its beginning:

```
> tables.move(a, 3, 6, 1);

> a:
[c, d, e, f, e, f, g, h]
```

See also: **move**, **purge**, **shift**, **swap**.

tables.new ([bool,] a, b [, k])

tables.new ([bool,] f, a, b [, k [, $\cdots$]])

tables.new (n, init = default)

In the first form, if no Boolean `bool` is given as the very first argument, the function creates a table array **[**a, a+k, $\cdots$, b-k, b**]**, with a, b, and k (the step size) being numbers. The step size is 1 if k - a number - is not given. If any Boolean `bool` is given as the very first argument, the function generates a linearly spaced table array of k numbers in the interval [a, b].

In the second form, if no Boolean `bool` is given as the very first argument, the function returns a table array **[**1~f(a), 2~f(a+k), $\cdots$, ((b-a)*1/k+1)~f(b)**]**, with f a function, a and b numbers. Thus, the function f is applied to all numbers between and including a and b. If f requires two or more arguments, the second, third, etc. argument must be passed after k. If any Boolean `bool` is given as the very first argument, the function generates a linearly spaced table array of k numbers in the interval [a, b] with f applied to all its members.

The function uses the Kahan-Babuška summation algorithm to prevent round-off errors in case the step size is non-integral.

In the third form, creates a table array of n slots, pre-filled with `default` which may be of any type.

Examples:

```
> tables.new(<< x, y -> x:x^2 + y >>, 1, 5, 1, 10):
[1:11, 2:14, 3:19, 4:26, 5:35]

> p := [0.1, 0.2, 0.1, 0.3, 1]
```

```
> tables.new( << x -> x:p[x] >>, 1, size p):
[1:0.1, 2:0.2, 3:0.1, 4:0.3, 5:1]

> tables.new(true, -4, 4, 6):
[-4, -2.4, -0.8, 0.8, 2.4, 4]

> tables.new(8, init = 0):
[0, 0, 0, 0, 0, 0, 0, 0]
```

tables.new also accepts functions that may return **null**. Example:

```
> tables.new(<< x -> if x % 3 = 0 then x else null fi >>, 0, 10):
[1 ~ 0, 4 ~ 3, 7 ~ 6, 10 ~ 9]
```

See also: **map**, **registers.new**, **sequences.new**.

tables.newtable (a, b)

Returns a table with `a` pre-allocated array slots and `b` pre-allocated hash slots. `a` and `b` should be non-negative integers. If `a` or `b` is negative, zero slots will be pre-allocated with no error being issued.

The function is useful only if you have to pass a table initialiser as a function argument, otherwise it is recommended to use the **create table** statement.

See also: **tables.dimension**, **create table/dict** statements.

tables.parts (t)

Returns both the array and the hash part of table `t` in two tables, with all key~value pairs preserved.

See also: **tables.array**, **tables.hash**.

tables.reshuffle (t)

The function moves all values in the hash part of table `t` to the end of its array part, thus emptying the hash part. The function works in-place, thus destructively, and returns no result.

See also: **sort**, **sorted**.

10.2 Sets

Summary of Functions:

Queries

 $$, **empty**, **filled**, **in**, **notin**, **size**, **type**, **typeof**.

Retrieving Values

 unpack.

Operations

 @, **$$**, **copy**, **map**, **remove**, **select**, **selectremove**.

Relational Operators

 =, ==, ~=, <>.

Cantor Operations

 intersect, **minus**, **subset**, **union**, **xsubset**.

Miscellaneous

 cleanse, **sets.newset**, **sets.resize**.

`cleanse (s)`

Empties set ₛ and returns the emptied structure. The memory previously occupied can be reused by the interpreter.

`copy (s)`

The function copies the entire contents of a set ₛ into a new set. See Chapter 8 for more information.

`empty (s)`

The operator checks whether a set ₛ does not contain any element. The return is **true** or **false**.

See also: **filled**.

filled (s)

The operator checks whether a set `s` contains at least one element. The return is **true** or **false**.

See also: **empty**.

map (f, s [, ···] [, true])

Maps the function `f` on all elements of a set `s`. See **map** in Chapter 8 for more information.

See also: **countitems**, **remove**, **select**, **selectremove**, **subs**, and **zip**.

remove (f, s [, ···] [, true])

Returns all values in set `s` that do not satisfy a condition determined by function `f`. See **remove** in Chapter 8 for more information.

See also: **map**, **select**, **selectremove**, **subs**, **zip**.

select (f, s [, ···] [, true])

Returns all values in set `s` that satisfy a condition determined by function `f`. See **select** in Chapter 8 for more information.

See also: **map**, **remove**, **selectremove**, **subs**, **zip**.

selectremove (f, s [, ···])

Returns all values in set `s` that satisfy and do not satisfy a condition determined by function `f`, in two sets. See **selectremove** in Chapter 8 for more information.

See also: **map**, **remove**, **select**, **subs**, **zip**.

size (s)

Returns the number of items in a set `s`.

typeof (s)

Returns the user-defined type assigned to set `s`.

The following functions have been built into the kernel as binary operators.

Please note that the operators returning a Boolean work in a Cantor way, i.e. `{1, 1}` `= {1}` → true, `{1, 2} xsubset {1, 1, 2, 2, 3, 3}` → true.

`s1 = s2`

This equality check of two sets `s1`, `s2` first tests whether `s1` and `s2` point to the same set reference in memory. If so, it returns **true** and quits.

If not, the operator then checks whether `s1` and `s2` contain the same items, and returns **true** or **false**. In this case, the search is linear.

`s1 == s2`

With sets, the `==` operator acts exactly as the `=` operator.

`s1 ~= s2`

With sets, the `~=` operator compares each element in `s1` and `s2` for approximate equality. See **approx** for further details. The return is either **true** or **false**.

`s1 <> s2`

This inequality check of two sets `s1`, `s2` first tests whether `s1` and `s2` do not point to the same set reference in memory. If so, it returns **true** and quits.

If not, the operator then checks whether `s1` and `s2` do not contain the same items, and returns **true** or **false**. In this case, the search is linear.

`c in s`

Checks whether the set `s` contains the item `c` and returns **true** or **false**. The search is constant.

`c notin s`

Checks whether the set `s` does not contain the item `c` and returns **true** or **false**. The search is constant.

`s1 intersect s2`

Searches all items in set `s1` that are also items in set `s2` and returns them in a set. The search is linear. If `t1` has a metatable and/or a user-defined type, then they will be copied to the result; otherwise the function will try to copy them from `t2`.

`s1 minus s2`

Searches all items in set `s1` that are not items in set `s2` and returns them as a set. The search is linear. If `s1` has a metatable and/or a user-defined type, then they will be copied to the result; otherwise the function will try to copy them from `s2`.

s1 `subset` s2

Checks whether all items in set `s1` are included in set `s2` and returns **true** or **false**. The operator also returns **true** if `s1` = `s2`. The search is linear.

s1 `union` s2

Concatenates two sets `s1` and `s2` simply by copying all its items to a new set. If `s1` has a metatable and/or a user-defined type, then they will be copied to the result; otherwise the function will try to copy them from `s2`.

s1 `xsubset` s2

Checks whether all items in set `s1` are included in set `s2` and whether `s2` contains at least one further item, so that the result is always **false** if `s1` = `s2`. The search is linear.

f `@` s

The operator maps a function `f` to all the values in set `s` and returns a set as the result. `f` must be a univariate function and return only one value. If `s` has metamethods or user-defined types, the return will also have them.

Examples:

```
> << x -> x^2 >> @ {1, 2, 3}:
{1, 4, 9}

> << x -> x > 1 >> @ {1, 2, 3}:
{false, true}
```

See also: **\$**, **\$\$**, **countitems**, **remove**, **select**, **selectremove**, **subs**, and **zip**.

f `\$` s

Returns all values in set `s` that satisfy a condition determined by function `f`. `f` should be a univariate function and return at least one value. In the multivariate case, all results but the first are ignored.

```
> << x -> x > 1 >> $  {1, 2, 3}:
{2, 3}
```

If present, the function also copies the metatable and user-defined type of `s` to the new set.

See also: **@**, **\$\$**, **map**, **remove**, **selectremove**, **subs**, **zip**.

f $$ s

Checks whether at least one element in set `s` satisfies the condition defined by function `f` and returns **true** or **false**. `f` should be a univariate function and return at least one value. In the multivariate case, all results but the first are ignored.

```
> << x -> x < 1 >> $$ {1, 2, 3}:
false
```

sets.newset (n)

Returns a set with `n` pre-allocated slots. `n` should be a non-negative integer.

The function is useful only if you have to pass a set initialiser as a function argument, otherwise it is recommended to use the **create set** statement.

sets.resize (s [, newsize [, true]])

Resizes set `s` to store at least `newsize` elements. If the last argument is **true** the number of pre-allocated slots will be adjusted to an optimum of the smallest power of 2 greater than or equal to n.

If only `s` is given, the number of pre-allocated slots will be changed to the smallest power of 2 greater than or equal the current size, usually freeing formerly occupied space.

If `newsize` < **size** `s` or the number of pre-allocated slots would not change, the function does nothing and returns without modifying the set.

The function returns the number of allocated elements and the number of pre-allocated slots.

See also: **math.nextpower**, **size**, **environ.attrib** maxsize and size values.

10.3 Sequences

Summary of Functions:

Queries

> **countitems**, **empty**, **filled**, **in**, **notin**, **size**, **typeof**.

Retrieving Values

> **getentry**, **unique**, **unpack**, **values**.

Operations

> **@**, **copy**, **copyadd**, **join**, **map**, **move**, **mulup**, **purge**, **qsumup**, **remove**, **reverse**, **select**, **selectremove**, **sumup**, **shift**, **sort**, **sorted**, **subs**, **swap**, **zip**.

Relational Operators

> **=, ==, ~=, <>**.

Cantor Operations

> **intersect**, **minus**, **subset**, **union**, **xsubset**.

With the exception of getentry**, map** and **zip**, the following functions have been built into the kernel as unary operators.

10.3.1 Operators

`copy (s)`

The function copies the entire contents of a sequence `s` into a new sequence. See Chapter 8 for more information.

`copyadd (s [, ···])`

See Chapter 8.

`countitems (item, s)`

`countitems (f, s [, ···])`

Counts the number of occurrences of an `item` in the sequence `s`. For further information, see Chapter 8.

`empty (s)`

The operator checks whether the sequence `s` does not contain any element. The return is **true** or **false**. See also: **filled**.

`filled (s)`

The operator checks whether the sequence `s` contains at least one element. The return is **true** or **false**. See also: **empty**.

`getentry (s [, k₁, ···, kₙ])`

Returns the entry $s[k_1, \cdots, k_n]$ from the sequence `s` without issuing an error if one of the given indices k_i (second to last argument) does not exist.

`join (s [, sep [, i [, j]]])`

Concatenates all string values in sequence `s` in sequential order and returns a string: `s[i]` & `sep` & `s[i+1]` ⋯ & `sep` & `s[j]`. The default value for `sep` is the empty string, the default for `i` is 1, and the default for `j` is the length of the sequence. The function issues an error if `s` contains non-strings.

Use the **tostring** function if you want to concatenate other values than strings, e.g.:

```
> join(map(tostring, seq(1, 2, 3))):
123
```

`map (f, s [, ···] [, true])`

Maps the function `f` on all elements of a sequence `s`. See **map** in Chapter 8 for more information. See also: **remove**, **select**, **subs**, **zip**.

`move (s1, start, stop, newidx [, s2])`

Copies elements from sequence `s1` to sequence `s2`, performing the equivalent to the following multiple assignment: `s2[newidx]`,⋯ = `s1[start]`, ⋯, `s1[stop]`. The default for `s2` is `s1`, i.e. elements are shifted in the same sequence. The destination range can overlap with the source range.

Returns the destination sequence `s2`.

See also: **purge**, **put**.

`mulup (s)`

Multiplies all numeric values in sequence `s`. See **mulup** in Chapter 8 for more information. See also: **sumup**.

`purge (s [, pos])`

`purge (s, a, b)`

In the first form, the function removes from sequence `s` the element at position `pos`, shifting down other elements to close the space, if necessary. Returns the value of the removed element, or nothing if `pos` is invalid. The default value for `pos` is n, where n is the length of the sequence, so that a call `purge(s)` removes the last element of `s`.

In the second form, it removes all elements starting from index `a` to index `b` (inclusive), moving excess elements down to close the space; the function automatically performs a garbage collection after shifting. In the 2nd form, nothing will be returned.

See also: **move**, **put**.

`qsumup (s)`

Raises all numeric values in sequence `s` to the power of 2 and sums up these powers. See **qsumup** in Chapter 8 for more information. See also: **sumup**.

`remove (f, s [, ···] [, true])`

Returns all values in sequence `s` that do not satisfy a condition determined by function `f`. See **remove** in Chapter 8 for more information. See also: **map**, **select**, **subs**, **zip**.

`reverse (s)`

Reverses the order of all elements in a sequence `s` in-place. The function returns the modified structure.

See also: **strings.reverse**, **stack.reversed**.

`select (f, s [, ···] [, true])`

Returns all values in sequence `s` that satisfy a condition determined by function `f`. See **select** in Chapter 8 for more information. See also: **map**, **remove**, **subs**, **zip**.

`selectremove (f, s [, ···])`

Returns all values in sequence `s` that satisfy and do not satisfy a condition determined by function `f`, in two sequences. See **selectremove** in Chapter 8 for more information. See also: **map**, **remove**, **select**, **subs**, **zip**.

`shift (s, a, b)`

Moves an element in sequence `s` from position `old` to `new`, with `old`, `new` integers, shifting all the other elements accordingly - which might also cause a rotation. The function returns nothing.

See also: **move**, **purge**, **swap**.

`size (s)`

Returns the number of items in a sequence `s`.

`sort (s [, comp])`

Sorts sequence `s` in a given order, and in-place. See **sort** in Chapter 8 for more information. See also: **sorted**, **skycrane.sorted**, **stats.issorted**, **stats.sorted**.

`sorted (s [, comp])`

Sorts sequence elements in `s` in a given order, but - unlike sort - not in-place, and non-destructively. See **sorted** in Chapter 8 for more information. See also: **sort**, **skycrane.sorted**, **stats.issorted**, **stats.sorted**.

`subs (x:v [, ···], s [, true])`

Substitutes all occurrences of the value `x` in sequence `s` with the value `v`. See **subs** in Chapter 8 for more information. See also: **map**, **remove**, **select**, **zip**.

`sumup (s)`

Sums up all numeric values in sequence `s`. See **sumup** in Chapter 8 for more information. See also: **qsumup**.

`swap (s, a, b)`

In sequence `s`, swaps the entries at index positions `a` and `b`, with `a`, `b` integers. The function returns nothing.

See also: **move**, **purge**, **shift**.

`typeof (s)`

Returns the user-defined type assigned to sequence `s`.

`unique (s)`

With a sequence `s`, the function removes multiple occurrences of the same item, if present in `s`. See **unique** in Chapter 8 for more information.

`values (s, i₁ [, i₂, ···]])`

Returns the elements from the given sequence `s` in a new sequence. This function is equivalent to

```
return seq( s[i₁], s[i₂], ··· )
```

See also: **ops**, **select**, **unpack**.

`zip (f, s1, s2)`

This function zips together two sequences `s1`, `s2` by applying the function `f` to each of its respective elements. See Chapter 8 for more information. See also: **map**, **remove**, **select**, **subs**.

See also: **bintersect**, **bisequal**, **bminus**, **purge**, **put** in Chapter 8 Basic Functions.

Following are the binary operators.

Please note that the operators returning a Boolean work in a Cantor way, i.e. `seq(1, 1) = seq(1)` → true, `seq(1, 2) xsubset seq(1, 1, 2, 2, 3, 3)` → true.

`s1 = s2`

This equality check of two sequences `s1`, `s2` first tests whether `s1` and `s2` point to the same sequence reference in memory. If so, it returns **true** and quits.

If not, the operator then checks whether `s1` and `s2` contain the same values without regard to their keys, and returns **true** or **false**. In this case, the search is quadratic.

`s1 == s2`

This strict equality check of two sequences `s1`, `s2` first tests whether `s1` and `s2` point to the same sequence reference in memory. If so, it returns **true** and quits.

If not, the operator then checks whether `s1` and `s2` contain the same number of elements and whether all entries in the sequences are the same and are in the same order, and returns **true** or **false**. In this case, the search is linear.

`s1 ~= s2`

This approximate equality check of two sequences `s1`, `s2` first tests whether `s1` and `s2` point to the same sequence reference in memory. If so, it returns **true** and quits.

If not, the operator then checks whether `s1` and `s2` contain the same number of elements and whether all entries in the sequences are approximately equal and are in the same order, and returns **true** or **false**. In this case, the search is linear. See **approx** for further information on the approximation check.

s1 <> s2

This inequality check of two sequences `s1`, `s2` first tests whether `s1` and `s2` do not point to the same sequence reference in memory. If so, it returns **true** and quits.

If not, the operator then checks whether `s1` and `s2` do not contain the same values, and returns **true** or **false**. In this case, the search is quadratic.

c in s

Checks whether the sequence `s` contains the value `c` and returns **true** or **false**. The search is linear. See also **binsearch** for binary search.

c notin s

Checks whether the sequence `s` does not contain the value `c` and returns **true** or **false**. The search is linear.

See also: **in** operator.

s1 intersect s2

Searches all values in sequence `s1` that are also values in sequence `s2` and returns them in a sequence. The search is quadratic. If `s1` has a metatable and/or a user-defined type, then they will be copied to the result; otherwise the function will try to copy them from `s2`.

s1 minus s2

Searches all values in sequence `s1` that are not values in sequence `s2` and returns them as a sequence. The search is quadratic. If `s1` has a metatable and/or a user-defined type, then they will be copied to the result; otherwise the function will try to copy them from `s2`.

s1 subset s2

Checks whether all values in sequence `s1` are included in sequence `s2` and returns **true** or **false**. The operator also returns **true** if `s1` = `s2`. The search is quadratic.

s1 union s2

Concatenates two sequences `s1` and `s2` simply by copying all its elements - even if they occur multiple times - to a new sequence.   If `s1` has a metatable and/or a user-defined type, then they will be copied to the result; otherwise the function will try to copy them from `s2`.

s1 xsubset s2

Checks whether all values in sequence `s1` are included in sequence `s2` and whether `s2` contains at least one further element, so that the result is always **false** if `s1` = `s2`. The search is quadratic.

f @ s

The operator maps a function `f` to all the values in sequence `s` and returns a sequence as the result. `f` must be a univariate function and return only one value. If `s` has metamethods or user-defined types, the return will also have them.

Examples:

```
> << x -> x^2 >> @ seq(1, 2, 3):
seq(1, 4, 9)

> << x -> x > 0 >> @ seq(1, 2, 3):
seq(true, true, true)
```

See also: **$** and **$$** operators.

f $ s

Returns all values in sequence `s` that satisfy a condition determined by function `f`. `f` should be a univariate function and return at least one value. In the multivariate case, all results but the first are ignored.

```
> << x -> x > 1 >> $ seq(1, 2, 3):
seq(2, 3)
```

If present, the function also copies the metatable and user-defined type of `obj` to the new sequence.

See also: @ operator, **countitems**, **descend**, **map**, **remove**, **selectremove, subs**, **unique**, **values**, **zip.**

f $$ s

Checks whether at least one element in sequence `s` satisfies the condition defined by function `f` and returns **true** or **false**. `f` should be a univariate function and return at least one value. In the multivariate case, all results but the first are ignored.

```
> << x -> x < 1 >> $$ seq(1, 2, 3):
false
```

See also: @ operator, **countitems**, **descend**, **map**, **remove**, **selectremove, subs**, **unique**, **values**, **zip.**

The following functions in the **base library** also support sequences:

Function	Meaning
bintersect	Same as the **intersect** operator but much faster with very large sequences.
bisequal	Same as the = operator but much faster with very large sequences.
bminus	Same as the **minus** operator but much faster with very large sequences.
duplicates	Returns all the values that are stored more than once in the given sequence.

10.3.2 sequences Library

This library provides generic functions for sequence manipulation. It provides all its functions inside the table `sequences`.

```
sequences.dimension (a:b [, c:d, ···] [, default])
sequences.dimension (a:b [, c:d, ···] [, init = default])
```

In the first form, creates a sequence of any dimension with index ranges `a:b` etc. with `a`, `b`, etc. integers, and an optional `default` for all its entries. `default` must not be a pair. The left-hand side values `a`, `c`, ... of the dimensions must always be 1.

In the second form the initialiser may be given as the option "init = `default`", which allows to also use pairs as a default.

If the initialiser is a structure, i.e. table, set, pair, sequence or register, then individual copies of the initialiser are created to avoid referencing to the same structure.

See also: **sequences.newtable**, **create sequence** statements.

```
sequences.new ([bool, ] a, b [, k])
sequences.new ([bool, ] f, a, b [, k [, ···]])
sequences.new (n, init = default)
```

In the first form, if no Boolean `bool` is given as the very first argument, the function creates a sequence **seq(**`a`, `a+k`, ···, `b-k`, `b`**)**, with `a`, `b`, and `k` (the step size) being numbers. The step size is 1 if `k` - a number - is not given. If any Boolean `bool` is given as the very first argument, the function generates a linearly spaced sequence of `k` numbers in the interval [`a`, `b`].

In the second form, if no Boolean `bool` is given as the very first argument, the function returns a sequence **seq(**$1\sim$`f(a)`, $2\sim$`f(a+k)`, ···, $((b-a)*1/k+1)\sim$`f(b)`**)**, with `f` a function, `a` and `b` numbers. Thus, the function `f` is applied to all numbers between and including `a` and `b`. If `f` requires two or more arguments, the second, third, etc. argument must be passed after `k`. If any Boolean `bool` is given as the very first

argument, the function generates a linearly spaced sequence of `k` numbers in the interval [`a`, `b`] with `f` applied to all its members.

The function uses the Kahan-Babuška summation algorithm to prevent round-off errors in case the step size is non-integral.

In the third form, creates a sequence of `n` slots, pre-filled with `default` which may be of any type.

Examples:

```
> sequences.new(<< x, y -> x:x^2 + y >>, 1, 5, 1, 10):
seq(1:11, 2:14, 3:19, 4:26, 5:35)

> p := seq(0.1, 0.2, 0.1, 0.3, 1)

> sequences.new( << x -> x:p[x] >>, 1, size p):
seq(1:0.1, 2:0.2, 3:0.1, 4:0.3, 5:1)

> sequences.new(true, -4, 4, 6):
seq(-4, -2.4, -0.8, 0.8, 2.4, 4)

> sequences.new(8, init = 0):
seq(0, 0, 0, 0, 0, 0, 0, 0)
```

sequences.new also accepts functions that may return **null**. In this case, an element is not added to the resulting structure. Example:

```
> sequences.new(<< x -> if x % 3 = 0 then x else null fi >>, 0, 10):
seq(0, 3, 6, 9)
```

See also: **map**, **tables.new**, **registers.new**.

`sequences.newseq (n)`

Returns a sequence with `n` pre-allocated slots. `n` should be a non-negative integer.

The function is useful only if you have to pass a sequence initialiser as a function argument, otherwise it is recommended to use the **create sequence** statement.

See also: **sequences.dimension**.

`sequences.resize (s [, newsize [, true]])`

Resizes sequence `s` to the given number of pre-allocated slots. If you actually shrink a sequence, then it discards any surplus elements.

The function returns the number of allocated elements and the number of pre-allocated slots, which may be vacant.

If `newsize` is 0 or `newsize` is less than the current size, then the function also purges all surplus values in the sequence.

If the optional third argument is **true** and `newsize` is non-zero, then the function sets the optimum number of pre-allocated slots to the smallest power of 2 greater than or equal to `newsize`.

If you pass just `s` without any further arguments, the function automatically allocates to the optimum number of slots without dropping any values.

Note that Agena automatically enlarges and shrinks a sequence if necessary when adding new or purging existing values, see **environ.kernel/seqautoshrink**.

See also: **math.nextpower**, **size**, **environ.attrib** maxsize and size values.

10.4 Registers

Summary of Functions:

Queries

countitems, **filled**, **in**, **size**.

Retrieving Values

getentry, **unique**, **unpack**, **values**.

Operations

copy, **copyadd**, **join**, **map**, **move**, **mulup**, **purge**, **remove**, **replace**, **sumup**, **select**, **selectremove**, **shift**, **sort**, **sorted**, **subs**, **swap**, **zip**.

Relational Operators

=, ==, ~=, <>.

Cantor Operations

intersect, **minus**, **subset**, **union**, **xsubset**.

With the exception of **getentry**, **map** and **zip**, the following functions have been built into the kernel as unary operators:

10.4.1 Operators

`copy (r)`

The function deep-copies the entire contents of a register `r` into a new register. See Chapter 8 for more information.

`copyadd (r [, ···])`

See Chapter 8.

`countitems (item, r)`

`countitems (f, r [, ···])`

Counts the number of occurrences of an `item` in the register `r`. For further information, see Chapter 8.

`duplicates (r [, option])`

Returns all the values that are stored more than once to the given register `r`, and returns them in a new register. Each duplicate will be returned only once.

If `option` is not given, the structure is sorted before evaluation since this is needed to determine all duplicates. The original structure is left untouched, however.

The total size of the new register is equal to the number of the elements in the result.

If a value of any type is given for `option`, the function assumes that the register has been already sorted. Otherwise it is suggested to use **skycrane.sorted** before the call to **duplicates** if the register contains values of different types, to prevent errors.

The function is written in Agena and included in the lib/library.agn file.

`empty (r)`

The operator checks whether the register `r` does not contain any element. The return is **true** or **false**. See also: **filled**.

`filled (r)`

The operator checks whether the register `r` contains at least one element. The return is **true** or **false**. See also: **empty**.

`getentry (r [, k₁, ···, kₙ])`

Returns the entry $r[k_1, \cdots, k_n]$ from the register `r` without issuing an error if one of the given indices k_i (second to last argument) does not exist.

`join (r [, sep [, i [, j]]])`

Concatenates all string values in register `r` in sequential order and returns a string: `r[i]` & `sep` & `r[i+1]` ⋯ & `sep` & `r[j]`. The default value for `sep` is the empty string, the default for `i` is 1, and the default for `j` is the top of the register. The function issues an error if `s` contains non-strings.

`map (f, r [, ···])`

Maps the function `f` on all elements of a register `r`. See **map** in Chapter 8 for more information.

See also: **has**, **remove**, **select**, **subs**, **zip**.

`move (r1, start, stop, newidx [, r2])`

Copies elements from register `r1` to register `r2`, performing the equivalent to the following multiple assignment: `r2[newidx],··· = r1[start], ···, r1[stop]`. The default for `r2` is `r1`, i.e. elements are shifted in the same register. The destination range can overlap with the source range.

Returns the destination register `r2`.

See also: **purge**, **put**, **swap**.

`mulup (r)`

Multiplies all numeric values in register `r`. See **mulup** in Chapter 8 for more information. See also: **sumup**.

`purge (r [, pos])`

`purge (r, a, b)`

In the first form, the function removes from register `r` the element at position `pos`, shifting down other elements to close the space, if necessary. Returns the value of the removed element, or nothing if `pos` is invalid. The default value for `pos` is n, where n is the length of the register, so that a call `purge(r)` removes the last element of `r`.

In the second form, removes all elements starting from index `a` to index `b` (inclusive), moving excess elements down to close the space; the function automatically performs a garbage collection after shifting. In the 2nd form, nothing will be returned.

Note that the function also reduces the top pointer of `r` by the number of elements removed.

See also: **move**, **put**, **shift**, **swap**.

`qsumup (r)`

Raises all numeric values in register `r` to the power of 2 and sums up these powers. See **qsumup** in Chapter 8 for more information. See also: **sumup**.

`remove (f, r [, ···] [, true])`

Returns all values in register `r` that do not satisfy a condition determined by function `f`. The total size of the new register is equal to the number of the elements in the result. See **remove** in Chapter 8 for more information. See also: **map**, **select**, **subs**, **zip**.

`select (f, r [, ···] [, true])`

Returns all values in register `r` that satisfy a condition determined by function `f`. The total size of the new register is equal to the number of the elements in the result. See **select** in Chapter 8 for more information. See also: **map**, **remove**, **subs**, **zip**.

`selectremove (f, r [, ···])`

Returns all values in register `r` that satisfy and do not satisfy a condition determined by function `f`, in two new registers. The total size of the new registers is equal to the number of the elements in the respective results. See **selectremove** in Chapter 8 for more information.

See also: **map**, **remove**, **select**, **subs**, **zip**.

`shift (r, a, b)`

Moves an element in register `r` from position `old` to `new`, with `old`, `new` integers, shifting all the other elements accordingly - which might also cause a rotation. The function returns nothing.

See also: **move**, **purge**, **swap**.

`size (r)`

Returns the total number of items assignable in register `r`.

`sort (r [, comp])`

Sorts register `r` in a given order, and in-place. All the values in the register up to the position pointed to by **the size** operator must be of the same type and non-**null**. See **sort** in Chapter 8 for more information. See also: **sorted**.

`sorted (r [, comp])`

Sorts register elements in `r` in a given order, but - unlike sort - not in-place, and non-destructively. All the values in the register up to the position pointed to by the **size** operator must be of the same type and non-**null**. See **sorted** in Chapter 8 for more information.

See also: **sort**.

`subs (x:v [, ···], r)`

Substitutes all occurrences of the value `x` in register `r` with the value `v`. See **subs** in Chapter 8 for more information.

See also: **map**, **remove**, **select**, **zip**.

`sumup (r)`

Sums up all numeric values in register `r`. See **sumup** in Chapter 8 for more information. See also: **qsumup**.

`swap (r, a, b)`

In registers `r`, swaps the entries at index positions `a` and `b`, with `a`, `b` integers. The function returns nothing.

See also: **move**, **purge**.

`unique (r)`

With a register `r`, the **unique** function removes multiple occurrences of the same item, if present in `r`, and returns a new register. The total size of the new register is equal to the number of the elements in the result. See **unique** in Chapter 8 for more information.

`values (r, i₁ [, i₂, ···]])`

Returns the elements from the given register `r` in a new register. This function is equivalent to

```
return reg( r[i₁], r[i₂], ··· )
```

The total size of the new register is equal to the number of the elements in the result. See also: **ops**, **select**, **unpack**.

`zip (f, r1, r2)`

This function zips together two registers `r1`, `r2` by applying the function `f` to each of its respective elements. See Chapter 8 for more information. See also: **map**, **remove**, **select**, **subs**.

The following functions have been built into the kernel as binary operators.

Please note that the operators returning a Boolean work in a Cantor way, i.e. `reg(1, 1) = reg(1)` → true, `reg(1, 2) xsubset reg(1, 1, 2, 2, 3, 3)` → true.

`r1 = r2`

This equality check of two registers `r1`, `r2` first tests whether `r1` and `r2` point to the same register reference in memory. If so, it returns **true** and quits.

If not, the operator then checks whether `s1` and `s2` contain the same values without regard to their keys, and returns **true** or **false**. In this case, the search is quadratic.

`r1 == r2`

This strict equality check of two registers `r1`, `r2` first tests whether `r1` and `r2` point to the same register reference in memory. If so, it returns **true** and quits.

If not, the operator then checks whether `r1` and `r2` contain the same number of elements and whether all entries in the registers are the same and are in the same order, and returns **true** or **false**. In this case, the search is linear.

`r1 ~= r2`

This approximate equality check of two registers `r1`, `r2` first tests whether `r1` and `r2` point to the same register reference in memory. If so, it returns **true** and quits.

If not, the operator then checks whether `r1` and `r2` contain the same number of elements and whether all entries in the registers are approximately equal and are in the same order, and returns **true** or **false**. In this case, the search is linear. See **approx** for further information on the approximation check.

`r1 <> r2`

This inequality check of two registers `s1`, `s2` first tests whether `s1` and `s2` do not point to the same register reference in memory. If so, it returns **true** and quits.

If not, the operator then checks whether `s1` and `s2` do not contain the same values, and returns **true** or **false**. In this case, the search is quadratic.

`c in r`

Checks whether the register `s` contains the value `c` and returns **true** or **false**. The search is linear. See also **binsearch** for binary search.

`r1 intersect r2`

Searches all values in register `r1` that are also values in register `r2` and returns them in a new register. The search is quadratic. The total size of the new register is equal to the number of the elements in the result. If `r1` has a metatable and/or a user-defined type, then they will be copied to the result; otherwise the function will try to copy them from `r2`.

`r1 minus r2`

Searches all values in register `r1` that are not values in register `sr2` and returns them as a new register. The search is quadratic. The total size of the new register is equal to the number of the elements in the result. If `r1` has a metatable and/or a user-defined type, then they will be copied to the result; otherwise the function will try to copy them from `r2`.

r1 <u>subset</u> r2

Checks whether all values in register `r1` are included in register `r2` and returns **true** or **false**. The operator also returns **true** if `r1 = r2`. The search is quadratic. The total size of the new register is equal to the number of the elements in the result.

r1 <u>union</u> r2

Concatenates two registers `r1` and `r2` simply by copying all its elements - even if they occur multiple times - to a new register. The total size of the new register is equal to the number of the elements in the result. If `r1` has a metatable and/or a user-defined type, then they will be copied to the result; otherwise the function will try to copy them from `r2`.

r1 <u>xsubset</u> r2

Checks whether all values in register `r1` are included in register `r2` and whether `r2` contains at least one further element, so that the result is always **false** if `r1 = r2`. The search is quadratic. The total size of the new register is equal to the number of the elements in the result.

f <u>@</u> r

In the first form, the operator maps a function `f` to all the values in register `r`. `f` should be a univariate function and return only one value. The return is a register. If `r` has metamethods or user-defined types, the return will also have them.

Examples:

```
> << x -> x^2 >> @ reg(1, 2, 3):
reg(1, 4, 9)

> << x -> x > 1 >> @ reg(1, 2, 3):
reg(false, true, true)
```

See also: **@** and **$$** operators, **map**, **reduce**, **remove**, **select**, **subs**, **times**, **zip**.

f <u>$</u> r

Returns all values in register `r` that satisfy a condition determined by function `f`. `f` should be a univariate function and return at least one value. In the multivariate case, all results but the first are ignored.

```
> << x -> x > 1 >> $ reg(1, 2, 3):
[2, 3]
```

If present, the function also copies the metatable and user-defined type of `r` to the new register.

All values up to the current top pointer are evaluated, and the size of the returned register is equal to the number of the elements in the return.

See also: @ operator, **countitems**, **descend**, **map**, **remove**, **selectremove, subs**, **unique**, **values**, **zip.**

f $$ r

Checks whether at least one element in register `r` satisfies the condition defined by function `f` and returns **true** or **false**. `f` should be a univariate function and return at least one value. In the multivariate case, all results but the first are ignored.

```
> << x -> x < 1 >> $$ reg(1, 2, 3):
false
```

All values up to the current top pointer are evaluated.

See also: @ operator, **countitems**, **descend**, **map**, **remove**, **selectremove, subs**, **unique**, **values**, **zip.**

The following functions in the **base library** also support registers:

10.4.2 registers Library

This library provides generic functions for register manipulation. It provides all its functions inside the table `registers`.

registers.dimension (a:b [, c:d, ···] [, default])

registers.dimension (a:b [, c:d, ···] [, init = default])

In the first form, creates a register of any dimension with index ranges `a:b` etc. with `a`, `b`, etc. integers, and an optional `default` for all its entries. `default` must not be a pair. The left-hand side values `a`, `c`, ... of the dimensions must always be 1.

In the second form the initialiser may be given as the option "init = `default`", which allows to also use pairs as a default.

If the initialiser is a structure, i.e. table, set, pair, sequence or register, then individual copies of the initialiser are created to avoid referencing to the same structure.

See also: **registers.newtable**, **create register** statements.

registers.extend (r, n)

Extends the given register `r` to - and not *by* - the given number of elements. All the elements already residing in `r` are kept. If `n` is less or equal to the current top (see

size), the structure is left unchanged and **false** will be returned - otherwise returns **true**.

See also: **registers.reduce**.

```
registers.new ([bool, ] a, b [, k])
registers.new ([bool, ] f, a, b [, k [, ···]])
registers.new (n, init = default)
```

In the first form, if no Boolean `bool` is given as the very first argument, the function creates a register **reg(**a, a+k, ···, b-k, b**)**, with a, b, and k (the step size) being numbers. The step size is 1 if k - a number - is not given. If any Boolean `bool` is given as the very first argument, the function generates a linearly spaced register of k numbers in the interval [a, b].

In the second form, if no Boolean `bool` is given as the very first argument, the function returns a register **reg(**1~f(a), 2~f(a+k), ···, ((b-a)*1/k+1)~f(b)**)**, with f a function, a and b numbers. Thus, the function f is applied to all numbers between and including a and b. If f requires two or more arguments, the second, third, etc. argument must be passed after k. If any Boolean `bool` is given as the very first argument, the function generates a linearly spaced register of k numbers in the interval [a, b] with f applied to all its members.

The function uses the Kahan-Babuška summation algorithm to prevent round-off errors in case the step size is non-integral.

In the third form, creates a register of n slots, pre-filled with `default` which may be of any type.

Examples:

```
> registers.new(<< x, y -> x:x^2 + y >>, 1, 5, 1, 10):
reg(1:11, 2:14, 3:19, 4:26, 5:35)

> p := reg(0.1, 0.2, 0.1, 0.3, 1)

> registers.new( << x -> x:p[x] >>, 1, size p):
reg(1:0.1, 2:0.2, 3:0.1, 4:0.3, 5:1)

> registers.new(true, -4, 4, 6):
reg(-4, -2.4, -0.8, 0.8, 2.4, 4)

> registers.new(8, init = 0):
reg(0, 0, 0, 0, 0, 0, 0, 0)
```

registers.new also accepts functions that may return **null**. Example:

```
> registers.new(<< x -> if x % 3 = 0 then x else null fi >>, 0, 10):
reg(0, null, null, 3, null, null, 6, null, null, 9, null)
```

See also: **map**, **tables.new**, **sequences.new**.

`registers.newreg (n)`

Returns a register with `n` pre-allocated slots. `n` should be a non-negative integer.

The function is useful only if you have to pass a register initialiser as a function argument, otherwise it is recommended to use the **create register** statement.

See also: **registers.dimension**.

`registers.reduce (r, n)`

Reduces register `r` to - and not *by* - to the first `n` given number of elements. All the elements residing above are removed. If the current top pointer is greater than `n`, it is reset to `n`.

See also: **registers.extend**.

`registers.settop (r, n)`

Sets the current position of the pointer to the top of register `r` to the given position `n`, a non-negative integer. Values above this position cannot be altered by any functions and operators. It returns **true** on success, and **false** otherwise. If the return is **false**, the current position of the top pointer has not been changed.

See also: **size**.

10.5 Pairs

Summary of Functions:

Queries

in, **notin**, **left**, **right**, **size**, **type**, **typeof**.

Operations

copy, **map**.

Relational Operators

=, ==, ~=, <>.

The following functionality has been built into the kernel as unary operators:

`copy (p)`

The function deep-copies the entire contents of a pair `p` into a new pair.

`map (f, p [, ···])`

Maps the function `f` on both elements of a pair `p` and returns a new pair. See **map** in Chapter 8 for more information.

`size (p)`

Returns the number of items in a pair `p`, i.e. always returns 2.

`type (p)`

Returns the type of a pair `p`, i.e. the string 'pair'.

`typeof (p)`

Returns either the user-defined type of the pair `p`, or the basic type 'pair'.

The following functionality has been built into the kernel as binary operators.

p1 = p2

This equality check of two pairs `p1`, `p2` first tests whether `p1` and `p2` point to the same pair reference in memory. If so, it returns **true** and quits.

If not, the operator then checks whether the left-hand side of `p1` and the left-hand side of `p2` are equal, and the same with both right-hand sides, and returns **true** or **false**.

p1 == p2

With pairs, the == operator acts exactly as the = operator.

p1 ~= p2

With pairs, the ~= operator compares the left-hand side of `p1` and the left-hand side of `p2` for approximate equality, and the same with both right-hand sides. The return is either **true** or **false**. See **approx** for further details.

p1 <> p2

This inequality check of two pairs `p1`, `p2` first tests whether `p1` and `p2` do not point to the same set reference in memory. If so, it returns **true** and quits.

If not, the operator then checks whether `p1` and `p2` do not contain the same items, and returns **true** or **false**.

c in p

Checks whether the number `c` fits into the closed interval with borders denoted by the numeric elements of pair `p`, and returns **true** or **false**.

c notin p

Checks whether the number `c` does not fit into the closed interval with borders denoted by the numeric elements of pair `p`, and returns **true** or **false**.

10.6 numarray - Numeric C Arrays

As a *plus* package, the numarray package is not part of the standard distribution and must be activated with the **import** statement, i.e. `import numarray`.

10.6.1 Introduction

The numarray package implements arrays of the C data types of either double, unsigned char, unsigned 2-byte and signed or unsigned 4-byte integers. The unsigned char type also supports bit fields.

The arrays implemented by this package are called numarrays for short.

Since numbers stored to numarrays consume less space, numarrays may be useful if a large amount of numbers have to be processed, but the amount of random-access memory of your system is limited.

While any element in a sequence, for example, occupies 24 bytes of memory, a number in a numeric stack takes only eight bytes, and a character in a character stack only one byte.

Also, numarrays are useful to store binary data. Operations on numarrays, however, are usually slower than those on Agena's native structures: tables, pairs, sequences or registers - so you will trade speed for memory.

Internally, numarrays are userdata structures that also support various metamethods.

You can create numarrays, assign and read numbers, resize arrays, store them to binary files, and read from files.

Functions to convert arrays to Agena's native structures, and vice versa, are provided, as well.

To create an array of unsigned chars, use **numarray.uchar**, of signed 64-bit doubles use **numarray.double**, of 16-bit unsigned integers use **numarray.ushort**, of 32-bit unsigned integers use **numarray.uint32** and of 32-bit signed integers use **numarray.int32**. The number of entries to be stored must be given when calling these three procedures. When creating arrays, all slots are automatically filled with zeros. Array indices count from 1, not 0. You can pass negative indices to access values from the end of an array.

Agena's standard indexing functions save and read numbers. So, for example, a[1] := -1 stores the number -1 to index 1 of the array a. a[1] reads the value stored at index 1 of the array a. Alternatively, **numarray.setitem** and **numarray.getitem** save and read numbers, respectively. Furthermore, **numarray.include** (bulk-assigns) numbers.

Arrays can be shrunk or extended with the **numarray.resize** function.

Arrays can be converted to sequences and registers with **numarray.toseq** and **numarray.toreg**. **numarray.toarray** creates arrays from tables, sequences and registers.

Functions **numarray.setbit**, **numarray.getbit**, **numarray.iterate** support bit fields with unsigned char arrays.

numarray.whereis searches for numbers, and **numarray.iterate** can sequentially traverse arrays.

numarray.write writes the contents of any array to a binary file. **numarray.readuchars** reads a complete file of unsigned chars, **numarray.readushorts** of unsigned 2-byte integers, **numarray.readdoubles** of doubles, **numarray.readlongdoubles** of longdoubles (see **long** package) and **numarray.readintegers** of signed integers with only one call. To open and close these files, use **binio.open** and **binio.close**. Most other binio function, such as **binio.sync**, **binio.rewind**, and **binio.filepos**, is supported, as well, with the exception of the **binio.read*** procedures. The low-level **numarray.read** function is used by the above mentioned **numarray.read*** functions.

The following metamethods exist: standard read and write indexing (see above), **in** and **notin** operators, strict and approximate equality (=, ==, <>, ~=, ~<> operators), **size**, **zero**, **nonzero** and `tostring`. To easily add further metamethods, have a look at the end of the lib/numarray.agn source file.

The arrays can store status information or other data in a special registry table that is available at pseudo-index position 0. You can use the index metamethod or **numarray.getitem** to read from or write data into this table, e.g. `n[0]` or `numarray.getitem(n, 0)`.

10.6.2 Functions

`numarray.convert (f, a [,···])`

Same as **numarray.map**, but processes in-place: Maps a function `f` on each element in the numarray `a` and changes the entries accordingly, i.e. the array elements will be transformed from `a`[1] to `f`(`a`[1]), etc. `f` must always return a number.

`numarray.cycle (a [, i [, p, [, true]]])`

Like **numarry.iterate**, but cycles through the numarray `a`, restarting from the `i`-th element which is 1 by default. For arguments `p` and `true`, see **numarray.iterate**.

`numarray.double (n)`

Creates a numarray of (signed) doubles (C double) with the given number of entries `n`, with `n` an integer, and with each slot set to the number 0.

Initially, the number of elements can be zero or more, use **numarray.resize** to extend the array before assigning values.

See also: **numarray.int32**, **numarray.longdouble**, **numarray.uint32**, **numarray.uchar**, **numarray.ushort**.

`numarray.getitem (a, i [, n])`

With `a` any numarray, returns the value stored at `a[i]`, where `i`, the index, is an integer counting from 1. The function is provided to avoid the index metamethod overhead.

If `n` is given, then besides `a[i]`, the values `a[i+1]` ... `a[i + n – 1]` are also returned as additional results. The default for `n` is 1.

See also: **numarray.iterate**, **numarray.setitem**, **numarray.replicate**, **numarray.subarray**.

`numarray.getbit (a, i)`

Returns the bit at index position `i` of uchar array `a`. `i` starts from 1, the rightmost bit, not zero.

The return is either 0 or 1.

See also: **getbit**, **numarray.setbit**, **numarray.iterate**.

`numarray.getsize (a)`

Returns the number of slots used by the numarray.

`numarray.include (a, pos, b)`
`numarray.include (a, i, x)`

In the first form, copies all values in the numarray `b` into the numarray `a`, starting at index `pos` (a number) of a. The function returns nothing. Both numarrays must by of the same type: either be uchar, integer, or double arrays. See also **numarray.setitem**.

In the second form, inserts a new number `x` into an array. First, the array is enlarged by one slot, all values starting at position `i` (thus including the value already stored at `a[i]`) are pushed to open space and finally the number `x` is assigned to `a[i]`.

The function returns nothing.

`numarray.int32 (n)`

Creates a numarray of signed 4-byte integers (C int32_t) with the given number of entries `n`, with `n` an integer, and with each slot set to the number 0. Initially, the number of elements can be zero or more.

See also: **numarray.double**, **numarray.resize**, **numarray.uchar**, **numarray.uint32**, **numarray.ushort**.

`numarray.iterate (a [, i [, p, [, true]]])`

Returns an iterator function that when called returns the next value in the numarray userdata structure `a`, or **null** if there are no further entries in the structure.

If an index `i` is passed, the first call to the iterator function returns the `i`-th element in the numarray list and with subsequent calls, the respective elements after index `i`.

You may also pass a positive integer step `p` to the iterator function: If given, then in subsequent calls the `p`-th element after the respective current one will be returned, equivalent to giving an optional step size in numeric for loops.

Bit Fields can be iterated one after the other by passing the fourth argument, the Boolean value **true**. (You may set `i` and `p` to 1 each to traverse all bits.)

Example 1: C doubles

```
> import numarray

> a := numarray.double(3)

> for i to 3 do a[i] := i * Pi od

> f := numarray.iterate(a, 2):  # return all values starting with index 2
procedure(01CDC200)

> f():
6.2831853071796

> f():
9.4247779607694

> f():  # no more values in a
null
```

Example 2: Bit Fields

```
> import numarray as n

> a := n.uchar(1)

> for i to 8 do n.setbit(a, i, 1) od

> n.get(a, 1):
255
```

```
> f := numarray.iterate(a, 1, 1, true) # iterate each bit, from the right

> f():
1
```

(etc.)

See also: **numarray.cycle**.

`numarray.longdouble (n)`

Creates a numarray of (signed) longdoubles (C long double) with the given number of entries n, with n an integer, and with each slot set to the number 0. Check the **long** package for further information.

Initially, the number of elements can be zero or more, use **numarray.resize** to extend the array before assigning values.

See also: **numarray.int32**, **numarray.double**, **numarray.uint32**, **numarray.uchar**, **numarray.ushort**.

`numarray.map (f, a [,···] [, true])`

Maps a function f on each element in the numarray a and returns a new numarray with the mapped results, i.e. the new array includes the values f(a[1]), f(a[2]), etc. f must always return a number.

If the last argument is the option inplace=**true**, or the Boolean **true**, then the operation will be done in-place, modifying the original array, but saving memory. After completion, the function returns the modified array.

See also: **numarray.convert**, **numarray.subs**.

`numarray.purge (a, i [, bool])`

Removes the value stored at a[i], shifting down other elements to close the space, and by default reduces the size of the array by one slot. If the array already is of size 0, an error will be returned. The function returns the value deleted.

If bool is **false**, the size of the array is not reduced. Instead, the last entry of the array is set to 0. Use **numarray.resize** if you want to finally shrink the array to its new smaller size. Passing the **false** option may be useful to avoid memory re-allocation overhead when deleting a lot of values at one time.

See also: **numarray.include**, **numarray.setitem**.

`numarray.read (fh [, bufsize])`

Reads data from the file denoted by its filehandle `fh` and returns a numarray userdata structure of unsigned C chars.

The file must be opened before with **binio.open** and must finally be closed with **binio.close**.

In general, the function reads in a limited amount of bytes per call. If only `fh` is passed, the number of bytes read is determined by the **environ.kernel**('buffersize') setting, usually 512 bytes.

You can pass the second argument `bufsize`, a positive integer, to read less or more bytes. Passing the `bufsize` argument may also be necessary if your platform requires that an internal input buffer is aligned to a certain block size.

The function increments the file position thereafter so that the next bytes in the file can be read with a new call to **numarray.read**.

If the end of the file has been reached, or there is nothing to read at all, **null** will be returned.

In case of an error, it quits with the respective error. Use one of the following functions to read   an entire file with only one call: **numarray.readdoubles**, **numarray.readintegers**, **numarray.readuchars**.

`numarray.readdoubles (fh [, bufsize])`

Reads all the numeric data from the file denoted by its filehandle `fh` and returns a numarray of C doubles.

By default, the function internally uses an input buffer of **environ.kernel**('buffersize') bytes, but you may choose another setting by passing the `bufsize` option. When passing an alternative buffer size, the function however reads in the entire file with only one call, too.

The file must be opened before with **binio.open** and finally be closed with **binio.close**.

The function is written in Agena (see lib/numarray.agn).

`numarray.readintegers (fh [, bufsize])`

Reads all the numeric data from the file denoted by its filehandle `fh` and returns a numarray of C (signed) int32_t's.

By default, the function internally uses an input buffer of **environ.kernel**('buffersize') bytes, but you may choose another setting by passing the `bufsize` option. When passing an alternative buffer size, the function however reads in the entire file with only one call, too.

The file must be opened before with **binio.open** and finally be closed with **binio.close**.

The function is written in Agena (see lib/numarray.agn).

`numarray.readlongdoubles (fh [, bufsize])`

Reads all the numeric data from the file denoted by its filehandle `fh` and returns a numarray of C long doubles, see **long** package.

By default, the function internally uses an input buffer of **environ.kernel**('buffersize') bytes, but you may choose another setting by passing the `bufsize` option. When passing an alternative buffer size, the function reads in the entire file with only one call, too.

The file must be opened before with **binio.open** and finally be closed with **binio.close**.

The function is written in Agena (see lib/numarray.agn).

`numarray.readuchars (fh [, bufsize])`

Reads all the numeric data from the file denoted by its filehandle `fh` and returns a numarray of C unsigned chars.

By default, the function internally uses an input buffer of **environ.kernel**('buffersize') bytes, but you may choose another setting by passing the `bufsize` option. When passing an alternative buffer size, the function however reads in the entire file with only one call, as well.

The file must be opened before with **binio.open** and finally be closed with **binio.close**.

The function is written in Agena (see lib/numarray.agn).

`numarray.readuint32 (fh [, bufsize])`

Reads all the numeric data from the file denoted by its filehandle `fh` and returns a numarray of C unsigned uint32_t's.

By default, the function internally uses an input buffer of **environ.kernel**('buffersize') bytes, but you may choose another setting by passing the `bufsize` option. When passing an alternative buffer size, the function however reads in the entire file with only one call, too.

The file must be opened before with **binio.open** and finally be closed with **binio.close**.

The function is written in Agena (see lib/numarray.agn).

`numarray.readushorts (fh [, bufsize])`

Reads all the numeric data from the file denoted by its filehandle `fh` and returns a numarray of 16-bit C unsigned integers.

By default, the function internally uses an input buffer of **environ.kernel**('buffersize') bytes, but you may choose another setting by passing the `bufsize` option. When passing an alternative buffer size, the function however reads in the entire file with only one call, too.

The file must be opened before with **binio.open** and finally be closed with **binio.close**.

The function is written in Agena (see lib/numarray.agn).

`numarray.remove (f, a [, ···] [, true])`

Returns all values in numeric array `a` that do not satisfy a condition determined by function `f` and returns a new array, or **null** if the condition has not been satisfied at all.

If `f` has only one argument, then only the function and the array are passed.

```
> numarray.remove(<< x -> x > 1 >>, a);
```

If the function has more than one argument, then all arguments *except the first* are passed right after the name of `a`.

```
> numarray.remove(<< x, y -> x > y >>, a, 1):   # 1 for y
```

If the last argument is the option inplace=**true**, or the Boolean **true**, then the operation will be done in-place, modifying the original array if the given condition has been satisfied at least once, but saving memory. After completion, the function returns the modified array. If the result is **null**, then the array has not been changed.

See also: **numarray.map**, **numarray.satisfy**, **numarray.select**, **numarray.subs**.

`numarray.replicate (a)`

Copies the entire contents of numarray `a` into a new array and returns it.

See also: **numarray.getitem**, **numarray.subarray**.

`numarray.resize (a, n)`

The function re-sizes a numarray userdata structure `a` to the given number of entries `n`. Thus you can extend or shrink a numarray. When extending, the function fills the

new array slots with zeros, while existing values are preserved. An array can be reduced to zero entries, as well.

The function returns the new size, an integer.

`numarray.satisfy (f, a [,···])`

With any numarray `a`, checks each element by calling function `f` which should return **true** or **false**. If at least one element in `a` does not satisfy the condition checked by `f`, the result is **false**, and **true** otherwise.

`numarray.select (f, a [, ···] [, true])`

Returns all values in numeric array `a` that satisfy a condition determined by function `f` and returns a new array, or **null** if the condition has not been satisfied at all.

If `f` has only one argument, then only the function and the array are passed.

```
> numarray.select(<< x -> x > 1 >>, a);
```

If the function has more than one argument, then all arguments *except the first* are passed right after the name of `a`.

```
> numarray.select(<< x, y -> x > y >>, a, 1):   # 1 for y
```

If the last argument is the option inplace=**true**, or the Boolean **true**, then the operation will be done in-place, modifying the original array if the given condition has been satisfied at least once, but saving memory. After completion, the function returns the modified array. If the result is **null**, then the array has not been changed.

See also: **numarray.map**, **numarray.remove**, **numarray.satisfy**, **numarray.subs**.

`numarray.setbit (a, i, n)`

Sets bit `n` at index position `i` of unsigned char array `a`. `n` must be either 0 or 1. `i` starts from 1, the rightmost bit, not from position zero. The function returns nothing.

See also: **setbit**, **numarray.getbit**, **numarray.iterate**.

`numarray.setitem (a, i, v)`

With `a` any numarray, sets number `v` to `a[i]`, where `i`, the index, is an integer counting from 1. The function is provided to avoid the __index metamethod overhead.

See also: **numarray.include**, **numarray.purge**.

numarray.sort (a)

Sorts a numarray `a` in ascending order, in-place. The function returns nothing.

numarray.sorted (a)

Sorts a numarray `a` in ascending order, non-destructively, and returns a new array.

numarray.subarray (a, i, j)

With `a` any numarray, returns the subarray `a[i` **to** `j]`, where `i`, `j`, the indices, are integers counting from 1. The function is provided to avoid the index metamethod overhead.

See also: **numarray.getitem**.

numarray.subs (x:v [, ···], a [, true])

Substitutes all occurrences of the value `x` in the numarray `a` with the value `v`. More than one substitution pair can be given. The substitutions are performed sequentially and simultaneously starting with the first pair.

```
> numarray.subs(1:3, 2:4, [1, 2, -1]):
[3, 4, -1]
```

If the last argument is the option inplace=**true**, or the Boolean **true**, then the operation will be done in-place, modifying the original array, but saving memory. After completion, the function returns the modified array.

You can check numbers for approximate instead of strict equality by passing the new strict=**false** option.

See also: **numarray.map**, **numarray.remove**, **numarray.select**.

numarray.toarray (o [, option])

Writes all data in the table array, sequence or register `o` into a numarray and returns it. By default, a double array will be returned (option `'double'`); if the second argument `option` is the string `'uchar'`, an unsigned char array is created; if it is the string `'integer'`, an integer array will be returned.

If a value in `o` is not a number, zero is written to the array.

numarray.toreg (a)

Receives numarray `a` and converts it into a register of numbers, the return.

numarray.toseq (a)

Receives numarray `a` and converts it into a sequence of numbers, the return.

numarray.uchar (n)

Creates a numarray of unsigned 1-byte characters (C unsigned char) with the given number of entries `n`, with `n` an integer, and with each slot set to the number 0. Initially, the number of elements can be zero or more.

See also: **numarray.double**, **numarray.int32**, **numarray.uint32**, **numarray.ushort**, **numarray.resize**.

numarray.uint32 (n)

Creates a numarray of unsigned 4-byte integers (C uint32_t) with the given number of entries `n`, with `n` an integer, and with each slot set to the number 0. Initially, the number of elements can be zero or more.

See also: **numarray.double**, **numarray.resize**, **numarray.uchar**, **numarray.int32**, **numarray.readuint32**, **numarray.uint32**, **numarray.ushort**.

numarray.used (a)

Returns the estimated number of bytes consumed by the given array `a`.

numarray.ushort (n)

Creates a numarray of unsigned 2-byte integers (C int16_t) with the given number of entries `n`, with `n` an integer, and with each slot set to the number 0. Initially, the number of elements can be zero or more.

See also: **numarray.double**, **numarray.int32**, **numarray.resize**, **numarray.uchar**, **numarray.uint32**.

numarray.whereis (a, what [, pos [, eps]])

Returns the index for a given value `what` in the numarray `a`. By default, the search starts at the beginning of the array, but you may pass any valid position `pos` (a positive integer) to determine where to start the search. The return is the index position, a positive number, or **null** if `what` could not be found in `a`.

By default, the function checks for exact equality to detect the existence of a value. By passing the fourth argument `eps`, a non-negative number, the function also compares the values approximately with the given maximum deviation `eps`. See **approx** for more details.

The '`__in`' metamethod internally uses this function to check for the existence of values.

`numarray.write (fh, a [, pos, nvalues]])`

Writes unsigned chars, doubles, long doubles or integers stored in a numarray `a` to the file denoted by its numeric file handle `fh`. The file must be opened with **binio.open** and closed with **binio.close**.

The start position `pos` is 1 by default but can be changed to any other valid position in the numarray.

The number of values (not bytes !) `nvalues` to be written can be changed by passing an optional fourth argument, a positive number, and by default equals the total number of entries in `a`, so the function can be called only once to write the entire array. Passing the `nvalues` argument may also be necessary if your platform requires internal buffers to be aligned to a particular block size. Depending on the type of data stored in `a`, the function automatically computes the number of bytes to be written.

The function returns the index of the next start position (an integer) for a further call, to write the next bunch of data in `a`, or **null**, if the end of the array has been reached. When the function returns **null**, then it also automatically flushes all unwritten content to the file so that you do not have to call **binio.sync** manually if you want to read the file subsequently.

No further information is stored to the file created, so you always must know the type of data you want to read in later.

Example on how to write an entire array of 4,096 integers piece-by-piece:

```
> a := numarray.int32(4 * 1024);

> fd := binio.open('integer.bin');

> pos := 1;

> do  # write 1024 values per each call
>    pos := numarray.write(fd, a, pos, 1024)
> until pos = null;

> binio.close(fd);
```

Use **binio.sync** if you want to make sure that any unwritten content is physically written to the file when calling **numarray.write** multiple times on one array.

If you want to add data to the end of a file later on, pass the `'a'` option to **binio.open**.

10.6.3 Metamethods

Metamethod	Functionality
`'__index'`	read operation, e.g. n[p] or n[p **to** q], with p, q any valid indices
`'__writeindex'`	write operation, e.g. n[p] := value, with p any valid index
`'__size'`	**size** operator, number of elements in a numarray
`'__in'`	**in** operator
`'__notin'`	**notin** operator
`'__empty'`	**empty** operator
`'__filled'`	**filled** operator
`'__tostring'`	formatting for output at the console
`'__aeq'`	approximate equality ~= operator
`'__eq'`	equality operator =
`'__eeq'`	strict equality operator ==
`'__zero'`	**zero** operator
`'__nonzero'`	**nonzero** operator
`'__gc'`	garbage collection

10.7 llist - Linked Lists

As a *plus* package, the **llist** package is not part of the standard distribution and must be activated with the **import** statement, e.g. `import llist`.

10.7.1 Introduction and an Example

Tables and sequences are quite slow if you have to insert or delete a lot of elements during an operation, for with each insertion or deletion, objects have to be shifted upward or downward physically.

To avoid these costly operations, data can also be represented in containers, or `nodes`, where "[e]ach node contains two fields: a "data" field to store whatever element [...], and a "next" field which is a pointer used to link one node to the next node.[19]" For example, if you would like to insert a new element at position n, the address of the `next entry` of node n - 1 is changed to the address of the new node containing the element to be inserted, and the `next entry` in the new node is assigned the address of the node containing the original value at position n.

This speeds up write operations by dimensions; read operations, however, are slower, for the linked list has to be traversed linearly. However, linked lists as implemented in this package are around fifteen times faster even when conducting a read operation with each write operation.

Metamethods exist to support printing, indexing, and indexed assignments; the **size**, **in**, **notin**, **=**, and **~=** operators are also supported.

Linked lists can contain **null**s, i.e. putting **null** into the data field of a node does not delete this node from the chain.

Linked list can store status information or other data in a special registry table that is available at pseudo-index position 0. You can use the index metamethod or **llist.getitem** to read from or write data into this table. Examples:

```
> print(a[0]);                 # print contents of status table

> print(llist.getitem(a, 0));  # dito

> a[0].cursor := 16;           # assign 16 to status table key 'cursor'

> a[0, 'cursor'] := 16;        # dito
```

For an example of how to use linked lists, see Chapter 6.27.

Note that the linked list implemented in this package always knows about the position of the top and the bottom element - so read and write access to them is

---

[19] For an excellent introduction on implementing linked lists, see "Linked List Basics", Copyright © 1998-2001, Nick Parlante. This quote has been taken from his manual, page 4.

always O(1).

10.7.2 Functions

`llist.append (l, obj [, ···])`

Appends one or more elements `obj` which may be of any type, to the singly-linked list `l`, in sequential order. There is no return.

See also: **llist.prepend**, **llist.put**.

`llist.checkllist (l)`

Checks whether its argument is a singly-linked list and issues an error otherwise. The function returns nothing.

`llist.dump (l)`

Writes each element in the singly-linked list `l` to a sequence and then deletes it from the list. The linked list thereafter is completely empty and cannot be used any longer. It will be garbage collected later as soon as you delete the reference to it. The return is the sequence.

The function can be used in case available memory is insufficient.

See also: **llist.toseq**.

`llist.getitem (l, idx [, n])`

Returns the item at index `idx` of the singly-linked list `l`. If the index does not exist, the function returns **null**.

If `idx` is negative, the function returns the value stored at the `-idx`'s position counting from end of the list.

If `n` is given, then besides `a[idx]`, the values `a[idx + 1]` ... `a[idx + n - 1]` are also returned as additional results. The default is 1.

See also: **llist.setitem**.

`llist.iterate (l [, n [, p]])`

Returns an iterator function that when called returns the next value in the singly-linked list `l`, which might also be **null** if one or more **null**s are included in the linked list, or **null** if there are no more entries in the list. Also returns **null** if the linked list is empty.

If an index `n` is passed, the first call to the iterator function returns the `n`-th element in the list and with subsequent calls, the respective elements after index `n`.

You may also pass a non-negative integer `p` to the iterator function: In this case, the next consecutive `p` elements in the list are skipped before determining and returning a value.

Example: Since the iterator can return **null** even if the end of the list has not yet been reached, we use a counter:

```
> L := llist.list(1); llist.append(L, null); llist.append(L, 2);

> f := llist.iterate(L);

> c := 0;

> while c++ < size L do
>    print(f())
> od;
1
null
2
```

The function can also process ulists.

See also: **ipairs**.

`llist.list ([···])`

The function creates a new singly-linked list and optionally stores all of the given elements in it. The return is a userdata of user-type `'llist'`.

`llist.prepend (l, obj [, ···])`

Prepends an element `obj`, and optionally further elements, which may be of any type, to the singly-linked list `l`. There is no return.

See also: **llist.append**, **llist.put**.

`llist.purge (l [, n])`

The function removes the element at position `n` from the linked list `l`. All the successors of the element to be deleted are `shifted` downwards. The function returns the value deleted, but issues an error if there is no element (i.e. node) at index `n`.

If `idx` is negative, the function deletes the value stored at the `-idx`'s position counting from end of the list.

If `n` is not given, then the last, i.e. top node is deleted; this is equal to **llist.purge**(`l`, size `l`).

The function can also process ulists.

llist.put (l, n, obj)

The function inserts the given element `obj` into singly-linked list `l` at position `n`. The original element at position `n` is not deleted - it and all of its successors are `shifted` to open space. The function returns nothing, and issues an error if the index is out-of-range.

If `idx` is negative, the function inserts the value at the `-idx`'s position counting from end of the list.

The function can also process ulists.

See also: **llist.append**, **llist.prepend**.

llist.replicate (l)

The function creates a copy of the singly-linked list `l` and returns a new linked list. However, if an element in `l` is a structure, it is not deep-copied.

llist.setitem (l, idx, obj)

Stores `obj`, which may be of of any type, to position `idx` of the singly-linked list `l`, overwriting the existing value. If size `l` > `idx` + 1 and the index does not yet exist, the function simply quits without an error. If `idx` = size `l` + 1, then the call is equivalent to **llist.append**. The function returns nothing.

If `idx` is negative, the function sets value to the `-idx`'s position counting from the end of the list.

See also: **llist.getitem**.

llist.toseq (l)

The function creates a new sequence and copies all elements in the singly-linked list `l` into it, in sequential order. The return is the sequence. If there are no elements in `l`, an empty sequence will be returned. If the list includes **null**s, they are ignored.

llist.totable (l)

The function creates a new table and copies all elements in the singly-linked list `l` into it, in sequential order. The return is the table. If there are no elements in `l`, an empty table will be returned. If the list includes **null**s, the resulting table will contain holes.

10.7.3 Unrolled Singly-Linked Lists

The **llist** package also supports unrolled singly-linked lists. You will find the respective functions in the `package` table **ulist**. To use the functions, it suffices you have already invoked the **llist** package with `import llist`.

Unrolled singly-linked lists (ulists) internally consist of a singly-linked list storing sequences of the actual values in each of its nodes. Various administrative information - the current number of sequences (i.e. nodes), the current and the maximum number of values in each sequence - is stored in a `registry` table at pseudo-index 0. However, this internal structure is hidden from the user, and you can use the same indices as you would do when calling **llist** functions to read or write values.

Insert and delete operations on ulists are twenty times faster when compared to singly-linked lists, with only a small increase of memory consumption. Similarly, simple read and write operations are 15 times faster.

Note that contrary to lllists, ulists cannot store **null**. The ulist package provides the following metamethods:

Functionality	ulist metamethod	ulist alternative
Pretty printer	**print** function, colon utility	ulist.tostring(L)
Reading values	L[k]	ulist.getitem(L, k)
Saving values	L[k] := v, etc.	ulist.setitem(L, k, v)
Size	**size** operator	ulist.getsize(L)
Existence check	**in** operator	ulist.has(L, v)
Existence check	**notin** operator	ulist.hasnot(L, v)
Equality check	= operator	ulist.isequal(K, L)

The following **ulist** functions work like the **llist** functions of the same name, with the exception of **ulist.list**:

ulist.append (ul, obj [, ···])

The function works like **llist.append**.

It is written in Agena and included in the lib/llist.agn file.

ulist.checkulist (ul)

Checks whether its argument is a ulist and issues an error otherwise. The function returns nothing.

See also: **ulist.isulist**.

`ulist.dump (ul)`

The function works like **llist.dump**.

`ulist.getitem (ul, idx [n])`

The function works like **llist.getitem**.

`ulist.getllist (ul, node)`

Returns the sequence stored at `node` (a positive integer) of the underlying llist. If the node does not exist, the function returns **null**.

`ulist.getsize (ul)`

Returns the number of items in a ulist.

`ulist.has (ul, v)`

Checks whether the ulist contains item `v` and returns **true** or **false**.

`ulist.isulist (ul)`

Checks whether its argument is a ulist and returns **true** or **false**.

See also: **ulist.checkulist**.

`ulist.iterate (ul [, n [, p]])`

The function works like **llist.iterate**.

It is written in Agena and included in the lib/llist.agn file.

See also: **ipairs**.

`ulist.list (n [, fill])`

The function creates a new unrolled singly-linked list and internally uses sequences with a maximum size of `n` slots. The default for `n` is 128.

If the number `fill` is given, with $0 <$ `fill` < 1, each underlying sequence is filled to the given percentage before a new one is created. The default is 0.75 for 75 percent. Reasonable values for `fill` may range between 0.5 to 0.75.

`ulist.prepend (ul, obj [, ···])`

The function works like **llist.prepend**.

It is written in Agena and included in the lib/llist.agn file.

`ulist.purge (ul, n)`

The function works like **llist.purge**, but also returns the element deleted.

It is written in Agena and included in the lib/llist.agn file.

`ulist.put (ul, n, obj)`

The function works like **llist.put**.

It is written in Agena and included in the lib/llist.agn file.

`ulist.setitem (ul, idx, value)`

The function works like **llist.setitem**.

`ulist.sort (ul [, f])`

The function works like **sort** and returns nothing.

`ulist.swap (ul, i, j)`

Swaps the positions of `ul[i]` and `ul[j]` in-place. The function returns nothing.

`ulist.tostring (ul)`

Converts the contents of a ulist to a formatted string that can be output at the prompt.

It is written in Agena and included in the lib/llist.agn file.

`ulist.toseq (ul)`

The function works like **llist.totable** but returns a sequence instead of a table.

See also: **ulist.dump**.

`ulist.totable (ul)`

The function works like **llist.totable**.

10.7.4 Doubly-Linked Lists

Finally, the **llist** package features doubly-linked lists. Read and write access to elements in doubly-linked lists is twice as fast as for singly-linked lists.

You find the respective functions in the package table **dlist**. It suffices you have already invoked the **llist** package with `import llist`.

The functions implemented for doubly-linked lists have the same name, work the same and have the same syntax as those for singly-linked lists, which are available in package table **llist**. Just replace the prefix `llist` with `dlist`.

dlist.append (l, obj [, ···])

Appends one or more elements `obj` which may be of any type, to the doubly-linked list `l`, in sequential order. There is no return.

See also: **dlist.prepend**, **dlist.put**.

dlist.checkdlist (l)

Checks whether its argument is a doubly-linked list and issues an error otherwise. The function returns nothing.

dlist.dump (l)

Writes each element in the doubly-linked list `l` to a sequence and then deletes it from the list. The linked list thereafter is completely empty and cannot be used any longer. It will be garbage collected later as soon as you delete the reference to it. The return is the sequence.

The function can be used in case available memory is insufficient.

See also: **dlist.toseq**.

dlist.getitem (l, idx [, n])

Returns the item at index `idx` of the doubly-linked list `l`. If the index does not exist, the function returns **null**.

If `idx` is negative, the function returns the value stored at the `-idx`'s position counting from end of the list.

If `n` is given, then besides `a[idx]`, the values `a[idx + 1]` ... `a[idx + n – 1]` are also returned as additional results. The default is 1.

See also: **dlist.setitem**.

`dlist.iterate (l [, n [, p]])`

Returns an iterator function that when called returns the next value in the doubly-linked list `l`, which might also be **null** if one or more **null**s are included in the linked list, or **null** if there are no more entries in the list. Also returns **null** if the linked list is empty.

If an index `n` is passed, the first call to the iterator function returns the `n`-th element in the list and with subsequent calls, the respective elements after index `n`.
You may also pass a non-negative integer `p` to the iterator function: In this case, the next consecutive `p` elements in the list are skipped before determining and returning a value.

Example: Since the iterator can return **null** even if the end of the list has not yet been reached, we use a counter:

```
> L := dlist.list(1); dlist.append(L, null); dlist.append(L, 2);

> f := dlist.iterate(L);

> c := 0;

> while c++ < size L do
>    print(f())
> od;
1
null
2
```

See also: **ipairs**.

`dlist.list ([···])`

The function creates a new doubly-linked list and optionally stores all of the given elements in it. The return is a userdata of user-type `'dlist'`.

`dlist.prepend (l, obj [, ···])`

Prepends an element `obj`, and optionally further elements, which may be of any type, to the doubly-linked list `l`. There is no return.

See also: **dlist.append**, **dlist.put**.

`dlist.purge (l [, n])`

The function removes the element at position `n` from the doubly-linked list `l`. All the successors of the element to be deleted are `shifted` downwards. The function returns the value deleted, but issues an error if there is no element (i.e. node) at index `n`.

If `idx` is negative, the function deletes the value stored at the `-idx`'s position counting from end of the list.

If `n` is not given, then the last, i.e. top node is deleted; this is equal to **dlist.purge**(`l`, size `l`).

dlist.put (l, n, obj)

The function inserts the given element `obj` into doubly-linked list `l` at position `n`. The original element at position `n` is not deleted - it and all of its successors are `shifted` to open space. The function returns nothing, and issues an error if the index is out-of-range.

If `idx` is negative, the function inserts the value at the `-idx`'s position counting from end of the list.

See also: **dlist.append**, **dlist.prepend**.

dlist.replicate (l)

The function creates a copy of the doubly-linked list `l` and returns a new linked list. However, if an element in `l` is a structure, it is not deep-copied.

dlist.setitem (l, idx, obj)

Stores `obj`, which may be of of any type, to position `idx` of the doubly-linked list `l`, overwriting the existing value. If size `l` > `idx` + 1 and the index does not yet exist, the function simply quits without an error. If `idx` = size `l` + 1, then the call is equivalent to **dlist.append**. The function returns nothing.

If `idx` is negative, the function sets value to the `-idx`'s position counting from the end of the list.

See also: **dlist.getitem**.

dlist.toseq (l)

The function creates a new sequence and copies all elements in the doubly-linked list `l` into it, in sequential order. The return is the sequence. If there are no elements in `l`, an empty sequence will be returned. If the list includes **null**s, they are ignored.

dlist.totable (l)

The function creates a new table and copies all elements in the doubly-linked list `l` into it, in sequential order. The return is the table. If there are no elements in `l`, an empty table will be returned. If the list includes **null**s, the resulting table will contain holes.

10.8 bags - Mulitsets

As a *plus* package, the **bags** package is not part of the standard distribution and must be activated with the **import** statement, e.g. `import bags`.

10.8.1 Introduction and Examples

A bag, also called a multiset, is a kind of Cantor set that also stores the number of occurrence of each unique element.

Consider a bulk of orders of books where each order is reported individually. You may only want to know how many times a book has been sold, instead of storing each individual order (and maybe all its data) to finally count them. You may want to save space and perform the count immediately as soon as the order has been committed.

The package uses tables of the user-defined type 'bag' to implement multisets.

A sequence of orders might look like this:

```
> import bags;

> orders := seq(
>    'Programming in Lua', 'Moon Lander', 'Lost Moon',
>    'Programming in Lua', 'Moon Lander', 'Lost Moon',
>    'C von A bis Z');

> books := bags.bag(unpack(orders));

> books['Lost Moon']:
2
```

For a further order, just enter

```
> bags.include(books, 'Agena');

> books:
bag(Agena ~ 1, C von A bis Z ~ 1, Lost Moon ~ 2, Moon Lander ~ 2,
Programming in Lua ~ 2)
```

A customer has cancelled his previous orders:

```
> bags.remove(books, 'Agena'):

> books:
bag(C von A bis Z ~ 1, Lost Moon ~ 2, Moon Lander ~ 2, Programming in Lua ~
2)
```

10.8.2 Functions & Metamethods

The package also provides the following metamethods:

Metamethod	Functionality
'__index'	read operation, e.g. n[p], with p an index
'__writeindex'	write operation, e.g. n[p] := value, with p the index
'__size'	**size** operator, number of characters currently stored
'__in'	**in** operator
'__notin'	**notin** operator
'__empty'	**empty** operator
'__filled'	**filled** operator
'__union'	**union** operator
'__intersect'	**intersect** operator
'__minus'	**minus** operator
'__tostring'	formatting for output at the console
'__gc'	garbage collection

The functions provided by the package are:

bags.attrib (b)

Returns the number of occurrence of all unique elements in the bag `b` and also the accumulated number of all occurrences of these elements in it. For example, the multiset bag('Curiosity' ~ 2, 'Skycrane' ~ 1) results to 2, 3.

See also: **bags.getsize**.

bags.bag ([···])

The function creates a new bag and optionally stores all of the given elements in it.

See also: **sykcrane.bagtable**.

bags.bagtoset (b)

The function returns all of the unique elements in `b` as a set.

bags.getsize (b)

Returns the number of occurrence of all unique elements in the bag `b` . without the overhead of calling **bags.attrib**. For example, the multiset bag('Curiosity' ~ 2, 'Skycrane' ~ 1) results to 2.

See also: **bags.attrib**.

`bags.include (b, obj [, ···])`

The function inserts all of the given elements `obj`, etc. into bag `b`.

The function returns nothing.

See also: **bags.minclude**, **tables.include**.

`bags.minclude (b, obj)`

The function inserts all of the given elements in the sequence `obj` into bag `b`. The function should be used instead of **bags.include** if the number of elements to be inserted exceeds Agena's argument stack.

The function returns nothing.

See also: **bags.include**.

`bags.remove (b, obj [, ···])`

The function removes all of the given elements `obj`, etc. from bag `b`. If the number of counts of the removed element reaches 0, the element will be deleted from the bag.

The function returns nothing.

There are metamethods for conducting some sort of arbitrary Cantor set operations on bags. Try out the binary operators **union** (for union), **minus** for difference set, **intersect** for intersection, **in** and **notin** for searching an object, plus support of the **size**, **empty** and **filled** operators.

If you would like to iterate a bag, you can use conventional **for/in** loops, for example, using the bag in the previous chapter:

```
> for i, j in books do print(i, j) od
Programming in Lua      2
C von A bis Z    1
Lost Moon        2
Moon Lander      2
```

10.9 bimaps - Bi-directional Maps

As a *plus* package, the **bimaps** package is not part of the standard distribution and must be activated with the **import** statement, i.e. `import bimaps`.

10.9.1 Introduction and Examples

The **bimaps** package implements a bi-directional map through tables. It is intended to hold items, i and j, that have a 1-to-1 relationship and allows to look up item j from table i and look up i from table j.

Examples:

```
> import bimaps

> l, r := bimaps.bimap()

> l.foo := 1

> l.bar := 2

> l.spam := 'eggs'

> l:
[bar ~ 2, foo ~ 1, spam ~ eggs]

> r:
[1 ~ foo, 2 ~ bar, eggs ~ spam]

> l = r:
true
```

10.9.2 Functions and Metamethods

The functions are:

bimaps.bimap ([tbl])

Creates two tables representing a bi-directional map. You can initialise the bimap by passing an optional table `tbl` with pre-defined values.

bimaps.attrib (bm)

Returns administrative information for bimap `bm`, see **environ.attrib** for details.

bimaps.entries (bm)

Returns all entries in bimap `bm`, without invoking any metamethods.

See also: **bimaps.indices**.

`bimaps.indices (bm)`

Returns all indices in bimap `bm`, without invoking any metamethods.

See also: **bimaps.entries**.

`bimaps.rawget (bm [, k])`

Returns the underlying table in bimap `bm`, without invoking any metamethods, if no index `k` is given - or if `k` is given, returns entry `bm[k]` without invoking any metamethods.

The package also provides the following metamethods:

Metamethod	Functionality
`'__index'`	read operation, e.g. n[p], with p any valid table index
`'__writeindex'`	write operation, e.g. n[p] := value, with p any valid index
`'__size'`	**size** operator, number of bi-directional pairs
`'__in'`	**in** operator
`'__notin'`	**notin** operator
`'__empty'`	**empty** operator
`'__filled'`	**filled** operator
`'__tostring'`	formatting for output at the console

10.10 heaps - Priority Queues

As a *plus* package, the **heaps** package is not part of the standard distribution and must be activated with the **import** statement, i.e. `import heaps`.

10.10.1 Introduction and Examples

The package implements skew heaps, `emulated` binary heaps and AVL trees which for example can be used as priority queues. A skew heap is a mostly unbalanced binary tree, usually avoiding costly reshuffles with each insert, whereas binary heaps and AVL trees are usually balanced - with extra cost at insertion.

The exact amortized complexity O(n) of all operations on a skew heap is known to be log(n, Phi), and the one for AVL and binary heaps is log(n, 2).

The package provides constructors (**avl.new**, **skew.new**, **binary.new**), metamethods (see table below), functions to insert new and replace existing values (**avl.include**, **skew.include**, **binary.include**), to remove the entry with the smallest index (**avl.remove**, **skew.remove**, **binary.remove**), and iterators (**avl.iterate**, **skew.iterate**, **binary.iterate**) that traverse the heaps in an ordered fashion - depending on the sorting method chosen at heap creation, which by default is in ascending order of the indices.

Usage is:

```
> import heaps;
```

All the functions to work with skew heaps reside in the `skew` table, those for binary heaps are in table `binary`, and those for AVL trees in table `avl`.

The package operations on binary heaps and AVL trees are at least 25 times faster than on skew heaps.

```
> h := binary.new()

> binary.include(h, 2, 'world')
> binary.include(h, 1, 'hello')
> binary.include(h, 10, 'everybody')

> k1, v1 := binary.remove(h)
> k2, v2 := binary.remove(h)
> k3, v3 := binary.remove(h)

> print(v1, v2, v3)
hello    world    everybody

> binary.include(h, 2, "world"); binary.include(h, 1, "hello");
> binary.include(h, 10, "everybody");

> f := binary.iterate(h);
```

```
> f():
1       hello

> f():
2       world

> f():
10      everybody
```

The skew heap functions have the same syntax and work the same.

10.10.2 Metamethods

The package provides the following metamethods for all three heap types:

Metamethod	Functionality
`'__index'`	read operation, e.g. n[p], with p any valid non-null index
`'__writeindex'`	skew heaps and AVL trees only: write operation, e.g. n[p] := value, with p any valid non-null index
`'__size'`	**size** operator, number of key~value pairs in the heap
`'__in'`	**in** operator
`'__notin'`	**notin** operator
`'__empty'`	**empty** operator
`'__filled'`	**filled** operator

10.10.3 Binary Heap Functions

binary.entries (h)

The function returns all entries in heap `h` in a new table. For the ordering, see **binary.iterate**.

See also: **binary.indices**.

binary.include (h, k, v)

Inserts a new key~value pair into heap `h`. The key `k` and value `v` must be non-null. The function returns nothing.

See also: **binary.remove**.

binary.indices (h)

The function returns all indices in heap `h` in a new table. For the ordering, see **binary.iterate**.

See also: **binary.entries**.

`binary.iterate (h)`

The factory returns an iterator that with each call returns a key~value pair from heap `h`, in an ordered fashion.

The ordering is determined by the comparison function passed to **binary.new**, which by default is a `less than` comparison, so that the iterator returns the values in ascending order of its indices.

`binary.new ([comparison])`

Creates an empty binary heap with the default comparison method for various operations the `less than` relation, i.e.

$$<< k1, k2 \to k1 \text{ and type } k1 = \text{type } k2 \text{ and } k1 < k2 >>.$$

You might pass another comparison function to be used.

`binary.remove (h)`

Removes the key~value pair with the smallest key and returns the key and the value removed. If the key does not exist, the function just returns **null**.

See also: **binary.include**.

`binary.reorder (h)`

The function deletes all obsolete datasets in binary heap `h` - to free memory.

10.10.4 AVL Tree Functions

The AVL tree functions are:

`avl.attrib (h)`

The function returns administrative information about an AVL tree: maximum height (key `maxheight`), number of key~value pairs currently included (key `length`) and the balance factor (key `balancefactor`).

`avl.entries (h)`

The function returns all entries in the AVL tree `h` in a new table, in ascending order of its respective indices.

See also: **avl.indices**.

`avl.getmax (h)`

The function returns the largest key along with its associated value from AVL tree `h`. The function does not remove the data from `h`.

See also: **avl.getmin**, **getminmax**, **avl.getroot**.

`avl.getmin (h)`

The function returns the smallest key along with its associated value from AVL tree `h`. The function does not remove the data from `h`.

See also: **avl.getmax**, **avl.getminmax**, **avl.getroot**.

`avl.getminmax (h)`

The function returns the smallest and largest key along with their associated values from AVL tree `h`. The function does not remove the data from `h`.

See also: **avl.getmax**, **getminmax**, **avl.getroot**.

`avl.getroot (h)`

The function returns the key along with its associated value from the root node of AVL tree `h`. The function does not remove the data from `h`.

`avl.include (h, k, v)`

Inserts a new key~value pair into AVL tree `h`. The key `k` must be non-null; if `v` is **null** then the function will call **avs.remove**(`h`, `k`). The function returns nothing.

See also: **avl.remove**.

`avl.indices (h)`

The function returns all indices in AVL tree `h` in ascending order and returns them in a new table.

See also: **avl.entries**.

`avl.iterate (h)`

The factory returns an iterator that with each call returns a key~value pair from AVL tree `h`, in ascending order of its indices.

`avl.new ()`

Creates an empty AVL tree.

`avl.remove (h)`

`avl.remove (h, k)`

In the first form, removes the key~value pair with the smallest key and returns the key and the value removed.

In the second form, deletes the given key~value pair from the tree.

If the key does not exist or the tree is empty, the function just returns **null**.

See also: **avl.include**.

10.10.5 Skew Heap Functions

The skew heap functions are:

`skew.entries (h)`

The function returns all entries in heap `h` in a new table. For the ordering, see **skew.iterate**.

See also: **skew.indices**.

`skew.height (h, k)`

Returns the height of a key `k` in skew heap `h`, with 0 depicting that the key is in the root node.

`skew.include (h, k, v)`

Inserts a new key~value pair into the skew heap `h`. The key `k` and value `v` must be non-null. The function returns nothing.

See also: **skew.remove**.

`skew.indices (h)`

The function returns all indices in heap `h` in a new table. For the ordering, see **skew.iterate**.

See also: **skew.entries**.

`skew.iterate (h)`

The factory returns an iterator that with each call returns a key~value pair from heap `h`, in an ordered fashion.

The ordering is determined by the comparison function passed to **skew.new**, which by default is a `less than` comparison, so that the iterator returns the values in ascending order of its indices.

`skew.new ([comparison])`

Creates an empty skew heap with the default comparison method for various operations the `less than` relation, i.e.

$$<< k1, k2 -> k1 \text{ and type } k1 = \text{type } k2 \text{ and } k1 < k2 >>.$$

You might pass another comparison function to be used.

`skew.remove (h)`

Removes the key~value pair with the smallest key from heap `h` and returns the key and the value removed. If the key does not exist, the function just returns **null**.

See also: **skew.include**.

`skew.reorder (h)`

The function balances skew heap `h` internally by popping the node with the highest priority and then re-inserting it. This is just for maintenance, you do not have to run this function before executing any other package function.

10.11 bfield - Bit Fields

The package provides lean, low-level functions to work with memory-saving bit fields. The functions are generally faster than those implemented in the **memfile** package.

Typical usage:

Create a bit field of at least ten bits, which is internally rounded up to 16 bits, as 16 is a multiple of eight whereas ten is not:

```
> m := bfield.new(10);
```

The field is by default pre-filled with zeros. If you want to preset other values, like 255 to set all bits to 1, enter:

```
> m := bfield.new(10, 0xff);
```

The actual size of the field is:

```
> size m:
16
```

Get some bits:

```
> m[1], m[9]:
1       1
```

Set some bits to zero:

```
> m[1] := 0; m[9] := 0;

> m[1], m[9]:
0       0
```

The package provides the following metamethods:

Metamethod	Functionality
'__index'	read operation, e.g. n[p], with p an index counting from 1; reads a bit, not a byte
'__writeindex'	write operation, e.g. n[p] := value, with p the index, counting from 1; sets a bit, not a byte
'__size'	**size** operator, number of bits in the field
'__zero'	**zero** operator, checks whether all field bits are set to zero
'__tostring'	formatting for output at the console; returns binary representations
'__gc'	garbage collection

The bit field functions are:

`bfield.clearbit (bitfield, n)`

Clears a bit, i.e. sets absolute bit position `n` in the `bitfield` to 0. `n` counts from 1.

The function returns nothing.

See also: **bfield.flipbit**, **bfield.getbit**, **bfield.setbit**, **bfield.setbitto**, **bfield.setbyte**.

`bfield.flipbit (bitfield, n)`

Flips the bit stored at absolute bit position `n` in the `bitfield`: if the current bit is 1, it is set to 0, and vice versa. `n` counts from 1.

The return is the bit value after flipping, either 1 or 0.

See also: **bfield.clearbit**, **bfield.setbit**.

`bfield.getbit (bitfield, n)`

Returns the bit stored at absolute bit position `n` in the `bitfield`. `n` counts from 1.

The return is either 1 or 0.

See also: **bfield.clearbit**, **bfield.setbit**.

`bfield.getbyte (bitfield, pos [, option])`

From `bitfield`, returns the *byte* at position `pos`, with `pos` > 0. The return is an integer in the range [0, 255].

See also: **bfield.getbit**, **bfield.getbytes**.

`bfield.new (n, [, val])`

Creates a bit field of at least `n` bits. If val is not given, then every *byte* in the field is set to zero. If `val` - a non-negative integer preferably in the range [0, 255] - is given, then every *byte* in the field is filled with it.

The number of bits actually allocated is always a multiple of 8, i.e. the field is filled up to whole bytes.

See also: **bfield.resize**.

`bfield.resize (bitfield, n [, val])`

Resizes the `bitfield` to exactly `n` bits, with `n` > 0. It can grow or shrink a bit field and in the latter case preserves the remaining content. If the bit field is to be enlarged, the function fills the new space with zeros if the third argument `val` is not given, otherwise the added *bytes* are set to the non-negative integer `val`, which should be in the range [0, 255].

The size of the modified bit field is always a multiple of 8.

See also: **bfield.new**.

`bfield.setbit (bitfield, n)`

Sets absolute bit position `n` in the `bitfield` to 1. `n` counts from 1. To clear a bit, use **bfield.clearbit**.

The function returns nothing.

See also: **bfield.clearbit**, **bfield.flipbit**, **bfield.getbit**, **bfield.setbyte**.

`bfield.setbitto (bitfield, n, val)`

Sets absolute bit position `n` in the `bitfield` to 1 the `val`, a non-negative integer preferably in the range [0, 255]. `n` counts from 1.

The function returns nothing.

See also: **bfield.clearbit**, **bfield.getbit**, **bfield.setbyte**.

`bfield.setbyte (bitfield, pos, val)`

Sets `val`, a non-negative integer preferably in the range [0, 255] into `bitfield` at *byte* position pos, with `pos` > 0.

The function returns nothing.

See also: **bfield.getbyte**, **bfield.setbit**.

Chapter **Eleven**

Numbers

11 Numbers

11.1 Mathematical Functions

The mathematical operators and functions explained in this chapter work on both real numbers as well as complex numbers, except if indicated otherwise.

For the sake of speed, basic arithmetic functions have been implemented as operators, whereas all other mathematical functions are implemented as Agena library functions (implemented either in C or Agena). While functions can be overwritten with self-defined versions, operators cannot be overwritten.

Summary of Operators and Functions:

Basic Arithmetic Operators

   +, -, *, /, /*, &+, &-, &*, &/, &\, math.accu, math.fdim, math.koadd.

Relational Operators

   =, ==, <, >, <=, >=, <>, |, approx.

Integer Division

   \, %, drem, iqr, modf, symmod, math.cld, math.fld, math.modiv, math.modulus, math.nearmod.

Exponentiation

   ^, **, antilog2, antilog10, cube, exp, expx2, frexp, ldexp, math.expminusone, math.exp2, math.exp10, math.iscube, math.issquare, square, squareadd.

Roots

   cbrt, hypot, hypot2, hypot3, hypot4, invsqrt, proot, pytha, root, sqrt, math.isqrt, fastmath.hypotfast, fastmath.sqroot, fastmath.sqrtfast.

Logarithms

   ilog2, ln, log, log2, log10, math.ceillog2, fastmath.lbfast, math.lnplusone, math.xlnplusone.

Trigonometric Functions

cas, cos, cot, csc, sec, sin, tan, math.quadrant, math.sincos, fastmath.sincosfast, math.wrap.

Inverse Trigonometric Functions

arccos, arccsc, arccot, arcsec, arcsin, arctan, arctan2, math.arccosh, math.arcsinh, math.arctansh.

Hyperbolic Functions

cosh, coth, csch, sech, sinh, tanh.

Inverse Hyperbolic Functions

arccosh, arccsch, arccoth, arcsech, arcsinh, arctanh.

Sign

sign, signum, math.copysign, math.flipsign, math.mulsign, math.signbit.

Miscellaneous

erf, erfc, erfcx, erfi, inverf, inverfc, fma, sinc, cosc, tanc, math.fib, math.fibinv, math.gcd, math.isfib, math.lcm, math.max, math.min, math.rectangular, math.relerror, math.triangular, muladd.

Miscellaneous Complex Functions

argument, bea, conjugate, cosxx, flip, polar.

Gamma, etc.

beta, binomial, fact, gamma, lngamma, math.fall, math.lnfact, math.pochhammer.

Bessel Functions

besselj, bessely.

Rounding Functions

ceil, entier, int, mdf, round, xdf, math.rint.

Numbers

> frac, frexp, ++, --, +++, ---, math.compose, math.decompose, math.eps, math.epsilon, math.exponent, math.fraction, math.frexp, math.mantissa, math.ndigits, math.nextafter, math.nextmultiple, math.nextpower, math.nthdigit, math.tohex, math.uexponent, math.ulp.

Numeric Checks

> even, finite, float, in, infinite, inrange, isint, isnegative, isnegint, isnonneg, isnonnegint, isnonposint, isnumber, isnumeric, isposint, ispositive, nan, odd, math.fpclassify, math.isinfinity, math.isminuszero, math.isnormal, math.isordered, math.ispow2, math.issubnormal.

Range Reduction and Conversion

> abs, ||, heaviside, math.branch, math.chop, math.clip, math.lnabs, math.norm, math.normalise, math.piecewise, math.ramp, math.rectangular, math.rempio2, math.unitise, math.unitstep, math.wrap.

Random Numbers

> math.random, math.randomseed.

Bases and Conversion

> math.convertbase, math.norm, math.tobinary, math.todecimal, math.tohex, math.toradians, math.tosgesim.

Primes

> math.congruentprime, math.isprime, math.nextprime, math.prevprime.

Bitwise Operators, Bit and Byte Twiddling

> &&, ~~, ||, ^^, <<<, >>>, <<<<, >>>>, implies, nand, nor, xnor, xor, getbit, getbits, getnbits, setbit, setbits, setnbits, math.inttofpb, bytes.numhigh, bytes.numlow, bytes.gethigh, bytes.getlow, bytes.numwords, math.fpbtoint, bytes.leadzeros, bytes.leastsigbit, bytes.mostsigbit, bytes.onebits, bytes.reverse, bytes.setdouble, bytes.sethigh, bytes.setnumhigh, bytes.setlow, bytes.setnumlow, bytes.setnumwords, bytes.tobytes.

Boolean Operators

> and, implies, nand, nor, not, or, xnor, xor

11.1.1 Operators and Functions

x ± y

The operator adds two numbers; returns a number. Complex numbers are supported.

See also: **factory.count**, **math.accu**, **math.koadd**, **inc** operator in Chapter 4.6.8.

x − y

The operator subtracts two numbers; returns a number. Complex numbers are supported.

See also: **math.fdim**, **dec** operator in Chapter 4.6.8.

x * y

The operator multiplies two numbers; returns a number. Complex numbers and Booleans are supported. A Boolean operand represents 1 for **true**, and 0 for **false** or **fail**.

See also: **mul** operator in Chapter 4.6.8.

x / y

The operator divides two numbers; returns a number. Complex numbers are supported.

See also: **recip,math.cld**, **math.fld**, **div** operator in Chapter 4.6.8.

x \ y

The operator performs an integer division of two numbers, and returns a number. The integer division is defined as: $x \setminus y = \textbf{sign}(x) * \textbf{sign}(y) * \textbf{entier}(|\frac{x}{y}|)$.

See also: **%**, **/**, **iqr**, **math.cld**, **math.fld**, **intdiv** operator in Chapter 4.6.8.

x &+ y

The operator adds two signed or unsigned 32-bit numbers; returns a number. Complex numbers are supported, as well. By default, the operator internally calculates with unsigned 32-bit integers. You can change this to signed integers by calling **environ.kernel** with the **signedbits** option.

See also: **bytes.add32**, **factory.count**, **math.accu**, **math.koadd**.

x &- y

The operator subtracts two signed or unsigned 32-bit numbers; returns a number. Complex numbers are supported, as well. By default, the operator internally calculates with unsigned 32-bit integers. You can change this to signed integers by calling **environ.kernel** with the **signedbits** option.

See also: **bytes.sub32**, **math.fdim**.

x &* y

The operator multiplies two signed or unsigned 32-bit numbers; returns a number. Complex numbers are supported. By default, the operator internally calculates with unsigned 32-bit integers. You can change this to signed integers by calling **environ.kernel** with the **signedbits** option.

See also: **bytes.mul32**.

x &/ y

The operator divides two signed or unsigned 32-bit numbers; returns a number. Complex numbers are supported. By default, the operator internally calculates with unsigned 32-bit integers. You can change this to signed integers by calling **environ.kernel** with the **signedbits** option. See also: **bytes.div32**.

x *% y

The operator multiplies two numbers and divides the result by 100; returns a number, the percentage.

x /% y

The operator divides two numbers and multiplies the result by 100; returns a number, the ratio.

x %% y

The operator computes the percentage change from the number x to the number y and returns a number. It is equivalent to y **/%** x - 100.

x +% y

The operator adds the given percentage y to x.

x -% y

The operator subtracts the given percentage y from x.

`z roll r`

The binary operator rotates a two-dimensional vector, represented by the complex number `z`, through the angel `r` (given in radians) counterclockwise and returns the new complex number `z`*exp(I*`r`). To convert degrees to radians, multiply by Pi/180. If `z` is just a number, it is internally converted to the complex number `z` + 0*I.

See also: **conjugate**, **flip**.

`x % y`

The modulus operator conducts the operation `x` % `y` = `x` - **entier**($\frac{x}{y}$)*`y`. The return is always non-negative.

See also: ****, **drem**, **everyth**, **iqr**, **symmod**, **hashes.fibmod32**, **math.modinv**, **math.modulus**, **math.wrap**, **mod** operator in Chapter 4.6.8.

`x symmod y`

The symmetric modulus operator evaluates the remainder of a division `x`/`y` (with `x`, `y` two Agena numbers). The result has the same sign as the numerator `x`. Specifically, the return value is `x` - q * `y`, where q is the quotient `x`/`y`, rounded towards 0 to the next integer.

See also: **\\,%**, **drem**, **iqr**, **math.modinv**, **math.modulus**, **math.wrap**.

`x ^ y`

The operator performs an exponentiation of real or complex `x` with a rational power `y`. With numbers, if `x` is negative and `y` non-integral, it returns **undefined**.

See also: **^** operator, **antilog2**, **antilog10**, **proot**, **root**, **square**, **squareadd**.

`x ** y`

The operator exponentiates the real or complex number `x` with the integer power `y`. Depending on the platform and with small `y`, the operator is at least 50 % faster than the **^** operator. If `y` is **undefined** or ±**infinity**, **undefined** will be returned.

See also: **cube**, **square**, **squareadd**.

`z squareadd c`

For numeric or complex `z`, `c`, computes `z`^{2} + `c`, preventing round-off errors.

See also: ******, **fma**, **square**.

⊥x⊥

The operator computes the absolute value of the number or complex number x, i.e. abs(x). The return in both cases is a number.

See also: **|-**, **abs**, **cabs**, **calc.eucliddist**, **math.lnabs**, **math.fdim**.

x ⊥- y

The operator computes the absolute difference of the two numbers x and y, i.e. abs(x - y). The return is a number.

See also: **|-**, **abs**, **cabs**, **calc.eucliddist**, **math.fdim**.

x && y

Bitwise `and` operation on two numbers x and y. By default, the operator internally calculates with unsigned 32-bit integers. You can change this to signed integers by calling **environ.kernel** with the **signedbits** option. See also: **environ.kernel** in Chapter 14.2. See also: **bytes.and32**.

+++ x

Returns the next representable number larger than x. If given a variable, the operator does *not* change its value. See also: **---**, **math.nextafter**.

--- x

Returns the next representable number smaller than x. If given a variable, the operator does *not* change its value. See also: **+++**, **math.nextafter**.

~~ x

Bitwise 32-bit complementary operation on the number x, i.e. bitwise NOT, flipping all the bits representing x. The operator returns signed results only, regardless of the environ.kernel/signedbits setting. See also: **bytes.not32**.

x ⊥⊥ y

Bitwise `or` operation on two numbers x and y. By default, the operator internally calculates with unsigned 32-bit integers. You can change this to signed integers by calling **environ.kernel** with the **signedbits** option. See also: **environ.kernel** in Chapter 14.2.

See also: **bytes.or32**.

x $\underline{\text{^^}}$ y

Bitwise 32-bit `exclusive-or` operation on two numbers x and y. By default, the operator internally calculates with unsigned 32-bit integers. You can change this to signed integers by calling **environ.kernel** with the **signedbits** option. See also: **environ.kernel** in Chapter 14.2.

See also: **bytes.xor32**.

x $\underline{\text{<<<}}$ y

Bitwise left-shift operation (multiplication by 2, i.e. x **<<<** y = $x*2^y$). By default, the operator internally calculates with signed 32-bit integers. You can change this to signed integers by calling **environ.kernel** with the **signedbits** option. If y >= **environ.kernel**('nbits'), returns 0. Please note that the results may vary across platforms with overflows and that shift by negative y are undefined. Shift by zero is the identity shift.

See also: **>>>**, **environ.kernel**, **bytes.shift32**.

x $\underline{\text{>>>}}$ y

Bitwise right-shift operation (division by 2, i.e. x **>>>** y = $x/2^y$). The operator by default calculates with unsigned 32-bit integers internally. You can change this to signed integers by calling **environ.kernel** with the **signedbits** option. If y >= **environ.kernel**('nbits'), returns 0. Please note that the results may vary across platforms with overflows and that shift by negative y are undefined. However, if x is negative and y positive, an arithmetic right-shift is accomplished, thus preserving the sign of x. A shift by zero is the identity shift.

See also: **<<<**, **environ.kernel**, **bytes.shift32**.

x $\underline{\text{<<<<}}$ y

Returns the number x rotated a given number of bits y to the left. Internally it uses unsigned 32-bit integers by default. You can change this to signed integers by calling **environ.kernel** with the **signedbits** option.

See also: **>>>>**, **environ.kernel**, **bytes.rotate32**.

x $\underline{\text{>>>>}}$ y

Returns the number x rotated a given number of bits y to the right. Internally it uses unsigned 32-bit integers by default. You can change this to signed integers by calling **environ.kernel** with the **signedbits** option.

See also: **<<<<**, **environ.kernel**, **bytes.rotate32**

x <u>in</u> **y**

Checks whether the number x is part of the interval defined by the pair y consisting of two numbers. The operator returns **true** or **false**. For a much faster check, see **inrange** operator.

x ⊥ **y**

The operator compares two finite numbers x, y, determines whether x is less than y, x is exactly equal to y, or x is greater than y, and returns -1, 0, or 1 respectively.

If at least one of the operators is infinite or **undefined**, the function returns **undefined**.

The operator is twice as fast as **sign**. See also: ~|, **signum**.

To build a piece-wise function, for example the absolute function, you may enter:

```
> my_abs := proc(x) is
>    case x | 0
>       of -1 then return -x
>    else
>       return x
>    esac
> end;
```

x <u>~⊥</u> **y**

The operator compares two finite numbers x, y, determines whether x is approximately equal to y, x is less than y, or x is greater than y, and returns 0, -1, or 1 respectively. See also: | operator.

abs (z)

If z is a number, the **abs** operator returns the absolute value of z. With a complex number $z = x + I*y$, it returns the distance between it and the origin as a number, i.e. $\sqrt{x^2 + y^2}$.

See also: ||, |-, **argument**, **cabs**, **math.lnabs**, **polar**.

antilog2 (z)

The operator computes 2^z, i.e. 2 raised to the power of the number or complex number z.

See also: ^ and ****** operators, **antilog10**, **log2**.

432   11 Numbers

`antilog10 (z)`

The operator computes 10^z, i.e. 10 raised to the power of the number or complex number `z`.

See also: `^` and ****** operators, **antilog2**, **log10**.

`approx (x, y [, eps])`

Compares the two numbers or complex values `x` and `y` and checks whether they are approximately equal. If `eps` is omitted, **Eps** is used.

The algorithm uses a combination of simple distance measurement (`|x-y| ≤ eps`) suited for values `near` 0 and a simplified relative approximation algorithm developed by Donald H. Knuth suited for larger values (`|x-y| ≤ eps * max(|x|, |y|)`), that checks whether the relative error is bound to a given tolerance `eps`.

The function returns **true** if `x` and `y` are considered equal or **false** otherwise. If both `a` and `b` are **infinity**, the function returns **true**. The same applies to `a` and `b` being -**infinity** or **undefined**.

See also: **math.eps**, **math.epsilon**.

`arccos (x)`

Returns the inverse cosine operator (`x` in radians). Complex numbers are supported.

`arccosh (x)`

Returns the inverse hyperbolic cosine of `x` (in radians). The function is implemented in Agena and included in the lib/library.agn file.

`arccsc (x)`

Returns the inverse cosecant of `x` (in radians). The function works on both numbers and complex values. The function is implemented in Agena and included in the lib/library.agn file.

`arccsch (x)`

Returns the inverse hyperbolic cosecant of `x` (in radians). The function works on both numbers and complex values. The function is implemented in Agena and included in the lib/library.agn file.

`arccot (x)`

Returns the inverse cotangent of `x` (in radians). The function works on both numbers and complex values. The function is implemented in Agena and included in the lib/library.agn file.

arccoth (x)

Returns the inverse hyperbolic cotangent of x (in radians). The function works on both numbers and complex values.

arcsec (x)

Returns the inverse secant of x (in radians). The operator works on both numbers and complex values.

arcsech (x)

Returns the inverse hyperbolic secant of x (in radians). The function works on both numbers and complex values. The function is implemented in Agena and included in the lib/library.agn file.

arcsin (x)

Computes the inverse sine operator (in radians). Complex numbers are supported.

arcsinh (x)

Returns the inverse hyperbolic sine of x (in radians). The function works on both numbers and complex values. See also: **math.arcsinh**.

arctan (x)

Computes the inverse tangent operator (in radians). Complex numbers are supported. See also: **arctan2**.

arctan2 (y, x)

Returns the arc tangent of y/x (in radians), but uses the signs of both parameters to find the quadrant of the result. (It also handles correctly the case of y being zero.) x and y must be numbers or complex numbers. See also: **arctan**.

arctanh (x)

Returns the inverse hyperbolic tangent of x (in radians). The function works on both numbers and complex values. The function is implemented in Agena and included in the lib/library.agn file. See also: **math.arctanh**.

argument (z)

Returns the argument (the phase angle) of the complex value z in radians as a number. If z is a number, the function returns 0 if $z \geq 0$, and π otherwise.

See also: **abs**, **cabs**, **polar**.

`bea (z)`

The operator takes the complex number $z = x!y$ and returns the complex number **sin**(x)***sinh**(y) + I***cos**(x)***cosh**(y). This function may be mathematically useless, but it creates beautiful fractals. With numbers, it returns **undefined**.

See also: **cosxx**, **flip**.

`beta (x, y)`

Computes the Beta function. x and y are numbers or complex values. The return may be a number or complex value. The Beta function is defined as: Beta(x, y) = $\frac{\Gamma x * \Gamma y}{\Gamma(x+y)}$, with special treatment if x and y are integers.

`binomial (n, k)`

Returns the binomial coefficient $\binom{n}{k}$ as a number. n, k may also be negative integers, or floats of any sign.

`besselj (n, x)`

Returns the Bessel function of the first kind. The order is n given as the first argument, the argument x as the second argument. The return is a number. The function works on both numbers and complex values.

`bessely (n, x)`

Returns the Bessel function of the second kind. The order n is given as the first argument, the argument x as the second argument. The return is a number. The function works on both numbers and complex values.

`cabs (z [, option])`

If z is a number, the **cabs** function returns the absolute value of z as a number (default) or abs(z) + I*0 if any option is given.

If z is a complex number $z = x + I*y$, contrary to the **abs** operator, it returns the real and imaginary absolute value, i.e. $|x| + I * |y|$.

See also: **||**, **|-**, **abs**, **argument**, **polar.**

`cartesian (x, y)`

Returns a complex number z in Cartesian notation a + I*b for magnitude/modulus x and argument/phase angle y. x and y must be numbers. The result is equivalent to z = x * **cis**(y).

See also: **cis**, **polar**.

`cas (x)`

Returns the `casine` of the number or complex number x the efficient way, i.e. $\sin(x) + \cos(x) = \sqrt{2}\sin(x+\frac{\pi}{4})$. It is written in Agena and included in the lib/library.agn file.

`cbrt (x)`

Returns the cubic root of the number or complex number x. With complex x, it is equal to $x\char`^(1/3)$, but not to root(x, 3).

See also: $\char`^$ operator, **root**.

`ceil (x)`

The function rounds upwards to the nearest integer larger than or equal to the number or complex number x. See the **entier** operator for a function that rounds downwards to the nearest integer. For the definition of **ceil**, see **entier**.

See also: **entier**, **floor**, **int**, **round**, **math.rint**.

`cis (x)`

Returns the complex exponential function exp(I*x) = cos(x) + I*sin(x) for any real or complex argument x. It is around 33 % faster than the equivalent expression exp(I*x). Note the equality **abs**(x) * **cis**(**argument**(x)) = x.

See also: **polar**.

`conjugate (z)`

The operator returns the conjugate x-I*y of the complex value z=x+I*y. If z is of type number, it is simply returned.

See also: **flip**.

`cos (x)`

The operator returns the cosine of x (in radians). Complex numbers are supported.

See also: **math.cospi**, **math.sincos**.

`cosc (x)`

The function returns the un-normalised cardinal cosine of x (in radians), i.e. cos(x)/x, with cosc(0) = undefined. Complex numbers are supported.

See also: **math.rectangular**, **sinc**, **tanc.**

`cosh (x)`

The operator returns the hyperbolic cosine of x (in radians). Complex numbers are supported. See also: **sinh**, **tanh**, **math.sinhcosh**.

`cosxx (z)`

The operator takes the complex number $z = x!y$ and returns the complex number **cos**(x)***cosh**(y)+I***sin**(x)***sinh**(y), i.e. the imaginary part of the result had the wrong sign. It represents FRACTINT's buggy cos function till v16. This function may be mathematically useless, but it creates beautiful fractals. With the number z, it returns cos(z).

See also: **cos**, **bea**, **flip**.

`cot (x)`

Returns the cotangent -tan($\frac{\pi}{2}+x$) as a number (in radians). The function is implemented in Agena and included in the lib/library.agn file. The function works on both numbers and complex values.

`coth (x)`

Returns the hyperbolic cotangent $\frac{1}{\tanh(x)}$ as a number (in radians). The function is implemented in Agena and included in the lib/library.agn file. The function works on both numbers and complex values.

`csc (x)`

Returns the cosecant $\frac{1}{\sin(x)}$ as a number (in radians). The function is implemented in Agena and included in the lib/library.agn file. The function works on both numbers and complex values.

`csch (x)`

Returns the hyperbolic cosecant as a number (in radians). The function is implemented in Agena and included in the lib/library.agn file. The function works on both numbers and complex values.

`cube (x)`

The operator raises the number or complex number x to the power of 3. See also: ******, **^**, **square** operators.

`drem (x, y)`

Evaluates the remainder of an integer division x/y (with x, y two Agena numbers), but contrary to **symmod**, rounds the internal quotient x/y to the nearest integer instead of towards zero. The function actually is a wrapper to C's remainder.

See also: \,**%**, **iqr**, **modf**, **symmod**, **math.modinv**, **math.modulus**.

`entier (x)`

The operator rounds the number x downwards to the nearest integer. For complex x, the return is:

$$re = real(x) - entier(real(x)) \text{ and } im = imag(x) - entier(imag(x)),$$
$$\text{then } entier(x) = int(real(x)) + I*int(imag(x)) + X \text{ , where}$$

$$X = \begin{cases} 0 & \text{if } a+b < 1 \\ 1 & \text{if } a+b \geqslant 1 \land a \geqslant b \\ I & \text{if } a+b \geqslant 1 \land a < b \end{cases}$$

Also: **ceil**(x) = -**entier**$(-x)$. (With numbers, the function internally calls C's floor.)

See also: **ceil**, **frac**, **int**, **mdf**, **round**, **math.rint**.

`erf (x)`

Returns the error function of x. It is defined by $erf(x) = \dfrac{2}{\sqrt{\pi}} \int_{t=0}^{x} e^{-t^2}$. The function works on both numbers and complex values.

See also: **erfc**, **erfcx**, **erfi**, **inverf**.

`erfc (x)`

Returns the complementary error function of x, a number or complex value. It is defined by $erfc(x) = 1 - erf(x)$. The return is a number or complex value.

See also: **erf**, **erfcx**, **erfi**, **inverfc**.

`erfcx (x)`

Implements the Scaled Complementary Error Function $erfcx(x) = \textbf{exp}(x^2)*\textbf{erfc}(x)$, with x a number or complex number and - depending on the type of x - a numeric or complex result.

See also: **erfc**, **erfcx**, **erfi**, **inverfc**.

`erfi (z)`

Computes the imaginary error function erfi(z) = -I***erf**(I*z) for real or complex z. The type of return depends on the type of z.

See also: **erf**, **erfc**, **erfcx**, **inverfc**.

`even (x)`

Checks whether the number x is even. The operator returns **true** if x is even, and **false** otherwise. With non-integral numbers, the operator returns **false**. With the complex value x, the operator returns **fail**. See also: **odd**.

`exp (x)`

Exponential function; the operator returns the value e^x, with e Euler's number. Complex numbers are supported. See also: **antilog2**, **antilog10**, **cis**, **expx2**, **math.expminusone**, **math.exp2**, **math.exp10**.

`exp2 (x)`

Returns 2^x with x any (complex) number. See also: **^** operator, **exp**, **exp10**.

`exp10 (x)`

Returns 10^x with x any (complex) number. See also: **^** operator, **exp**, **exp2**.

`expx2 (x [, sign])`

Computes either e$^{x\text{\^{}}2}$ if `sign` $\geq$ 0, or e$^{-x\text{\^{}}2}$ if `sign` < 0 while suppressing error amplification that would occur from the in-exactness of the exponential argument x^2. x may be a number or complex number, while `sign` must be a number. By default, `sign` is positive.

`fact (n)`

Returns the factorial of n, i.e. the product of the values from 1 to n. n may be a non-negative integer or a negative or positive rational number that is non-integral, otherwise the function returns **undefined**. The function is implemented in Agena and included in the lib/library.agn file. It features a defaults remember table (rotable) which you may extend by adding new defaults to your `agena.ini` file (see **rtable.defaults** and Appendix A6).

See also: **math.fall**, **math.pochhammer**.

`finite (x)`

Checks whether the number or complex number x is neither ±**infinity** nor **undefined** (C NaN). The operator returns **true** or **false**.

See also: **even**, **float**, **infinite**, **nan**, **odd**, **math.isinfinity**, **math.isordered**.

`flip (z)`

The operator takes the complex number z and returns the new complex number **imag**(z)!**real**(z), i.e. the real and imaginary parts are swapped. With numbers, always returns 0.

See also: **bea**, **conjugate**, **cosxx**.

`floor (x)`

The function rounds downwards to the nearest integer larger than or equal to the number or complex number x. It works like the **entier** operator.

`float (x)`

Checks whether the number x is a float, i.e. not an integer, and returns **true** or **false**.

With complex numbers, if returns **true** if the real part is integral and the imaginary part is zero, and **false** otherwise.

If x is not a (complex) number, the operator returns **fail**. With +/-**infinity** and **undefined**, returns **false**.

See also: **finite**, **integral**, **isint**.

`fma (x, y, z)`

Performs the fused multiply-add operation `(x * y) + z`, with the intermediate result *not* rounded to the destination type, to improve the precision of a calculation. x, y, and z must be numbers or complex numbers. See also: **squareadd** and **muladd** operators.

`frac (x)`

Returns the fractional part of the number x, i.e. x – **int**(x), thus preserving the sign. With complex numbers a + I*b, returns **frac**(a) + I***frac**(b).

See also: **entier**, **int**, **modf**.

`frexp (x)`

Returns the mantissa m and the exponent e of the number x such that $x = m2^e$. e is an integer, and the value of m is in the range [0.5, 1) (or zero when x is zero). The operation is bijective, i.e. ldexp(frexp(x)) = x. With complex x, returns m and e both for the real and the imaginary part.

See also: **frexp10**, **ilog2**, **ldexp**, **math.exponent**, **math.frexp**, **math.mantissa**.

`frexp10 (x)`

Returns the mantissa m and the exponent e of the number x such that $x = m10^e$. e is an integer, and the value of m is in the range [0, 1). Since floats are represented with base 2, and not base 10, the operation is not bijective, i.e. **ldexp**(**frexp**(x)) ≠ x. With complex x, returns m and e both for the real and the imaginary part. See also: **frexp**.

`gamma (x)`

The gamma function Γ x. x may be a number or complex value.

See also: **lngamma**.

`heaviside (x [, z])`

The Heaviside function. Returns 0 if $x < 0$, 1 if $x > 0$, and z if $x = 0$, where z defaults to **undefined**. The function is implemented in Agena and included in the lib/library.agn file.

See also: **calc.smoothstep**, **math.clip**, **math.ramp**, **math.rectangular**, **math.unitise**, **math.unitstep**.

`hypot (x, y)`

Returns $\sqrt{x^2+y^2}$ with x, y numbers, complex numbers or a mix of them. With x, y numbers, it is the length of the hypotenuse of a right triangle with sides of length x and y, or the distance of the point (x, y) from the origin. The function is slower but more precise than using **sqrt** along with **square**, avoiding over- and underflows and treating subnormal numbers accordingly. The return is a number or complex number.

See also: **hypot2**, **hypot3**, **hypot4**, **invhypot**, **pytha**, **root**, **sqrt**, **calc.eucliddist**.

`hypot2 (x)`

Returns the number or complex number $\sqrt{1+x^2}$, with x a number or complex number. The function is slower but more precise than using **sqrt** along with **square**, avoiding over- and underflows and treating subnormal numbers accordingly.

See also: **hypot**, **hypot3**, **hypot4**, **root**, **sqrt**.

`hypot3 (x)`

Returns the number $\sqrt{1-x^2}$, with x a number or complex number. The function is slower but more precise than using **sqrt** along with **square**, avoiding over- and underflows and treating subnormal numbers accordingly.

See also: **hypot**, **hypot2**, **hypot4**, **root**, **sqrt**.

`hypot4 (x, y)`

Returns the number $\sqrt{x^2-y^2}$, with x, y numbers, complex numbers or a mix of them. The function is slower but more precise than using **sqrt** along with **square**, avoiding over- and underflows and treating subnormal numbers accordingly.

See also: **hypot**, **hypot2**, **hypot3**, **root**, **sqrt**.

`ilog2 (x)`

Extracts the exponent of the number or complex number x (i.e. the integer part of the base-2 logarithm of the positive number x) and returns it as the number **entier**(**log2**(x)).

See also: **frexp**, **ilog10**, **ln**, **log**, **log2**, **log10**, **math.ceillog2**, **math.ispow2**.

`ilog10 (x)`

Extracts the exponent of number x and returns it as the number **entier**(**log10**(x)).

See also: **ilog2**, **log10**.

`implies (x, y)`

With two booleans, the function computes **not**(x) **or** y, with numbers returns:

$$(\sim\sim x) \mathbin{||} y.$$

`infinite (x)`

Checks whether the number or complex number x is ±**infinity**. The operator returns **true** or **false**.

See also: **even**, **float**, **finite**, **nan**, **odd**, **math.isinfinity**, **math.isordered**.

`inrange (x, a, b)`

The operator checks whether x is part of the closed interval [a, b] and returns **true** or **false**. All arguments must be numbers.

See also: **in** operator.

`int (x)`

Rounds x to the nearest integer towards zero. The operator also supports complex numbers. To round a float to a given decimal place, use **xdf**. To get the fractional part of a number, call **frac**.

See also: \ operator, **ceil**, **entier**, **float**, **iqr**, **mdf**, **modf**, **round**, **math.rint**, **xdf**.

`integral (x)`

Checks whether the number x is an integer, i.e. not a float, and returns **true** or **false**.

With complex numbers, if returns **true** if the real part is integral and the imaginary part is zero, and **false** otherwise.

If x is not a (complex) number, the operator returns **fail**. With +/-**infinity** and **undefined**, returns **false**.

See also: **finite**, **float**, **isint**, **multiple**.

`inverf (x)`

Computes the inverse error function erf^{-1}(x), where x is a number.

See also: **erf**, **inverfc**.

`inverfc (x)`

Computes the inverse complimentary error function erfc^{-1}(x), where x is a number.

See also: **erfc**, **inverfc**.

`invhypot (x, y)`

Computes $1/\sqrt{x^2+y^2} = 1/$**hypot**(x, y), is 35 % faster than the naive 1/**hypot** approach and is protected against underflow and overflow.

See also: **hypot**.

`invsqrt (x)`

Returns the inverse square root of numeric or complex x, i.e. 1/**sqrt**(x).

See also: **sqrt**.

`iqr (x, y)`

Computes both the integer quotient and the integer remainder - rounded toward zero - of the number x divided by the number y and returns them. If x or y are not integers, the function returns **undefined** twice. The function is equivalent to the Agena representation:

```
iqr := proc(x :: number, y :: number) is
   if float(x) or float(y) then
      return undefined, undefined
   else
      return x \ y, x symmod y
   fi
end;
```

See also: \ and % operators, **drem**, **modf**, **math.cld**, **math.fld**, **math.modinv**, **math.modulus**, **symmod**.

`iscomplex (···)`

Checks whether the given arguments are all of type **complex** and returns **true** or **false**.

`isint (···)`

Checks whether all of the given arguments are integers and returns **true** or **false**. If at least one of its arguments is not a number, the function returns **fail**.

See also: **float**, **integral**.

`isnegative (···)`

Checks whether all of its arguments are negative numbers and returns **true** or **false**. If at least one of its arguments is not a number, the function returns **fail**.

See also: **isnegint**, **isnegative**, **isnonneg**, **ispositive**.

`isnegint (···)`

Checks whether all of the given arguments are negative integers and returns **true** or **false**. If at least one of its arguments is not a number, the function returns **fail**.

See also: **isnonnegint**, **isposint**, **isnegative**, **ispositive**.

`isnonneg (···)`

Checks whether all of its arguments are zero or positive numbers and returns **true** or **false**. If at least one of its arguments is not a number, the function returns **fail**.

See also: **isnegint**, **isposint**, **isnegative**, **ispositive**.

`isnonnegint (···)`

Checks whether all of the given arguments are zeros or positive integers and returns **true** or **false**. If at least one of its arguments is not a number, the function returns **fail**.

`isnonposint (···)`

Checks whether all of the given arguments are zeros or negative integers and returns **true** or **false**. If at least one of its arguments is not a number, the function returns **fail**.

`isnumber (···)`

Checks whether the given arguments are all of type **number** and returns **true** or **false**.

`isnumeric (···)`

Checks whether the given arguments are all of type **number** or of type **complex** and returns **true** or **false**.

See also: **numeric**.

`isposint (···)`

Checks whether all of its arguments are positive integers and returns **true** or **false**. If at least one of its arguments is not a number, the function returns **fail**.

See also: **isnonposint**.

`ispositive (···)`

Checks whether all of its arguments are positive numbers and returns **true** or **false**. If at least one of its arguments is not a number, the function returns **fail**.

See also: **isposint**, **isnegative**, **isnonneg**.

`ldexp (m, e)`

Returns m2^e (`e` should be an integer, and `m` must be a number).

See also: **frexp**.

`ln (x)`

Natural logarithm of `x` with the base e[1]. If `x` is non-positive, the operator returns **undefined**. Complex numbers are supported.

See also: **exp**, **log**, **log2**, **log10**.

`lngamma (x)`

Computes ln Γ `x`. If `x` is a non-positive number, the operator returns **undefined**. Complex numbers are supported.

See also: **gamma**, **calc.Psi**.

`log (x, b)`

The operator returns the logarithm of the number or complex number `x` to the base `b`, with `b` a number or a complex number.

See also: **ln**, **log2**, **log10**.

`log2 (x)`

Returns the base-2 logarithm of the number or complex number `x`.

See also: **antilog2**, **ilog2**, **ln**, **log**, **log10**, **math.ceillog2**.

`log10 (x)`

Returns the base-10 logarithm of the number or complex number `x`.

See also: **antilog2**, **ilog10**, **ln**, **log**, **log2**.

`mdf (x [, n])`

Rounds up the number x at its n-th decimal place and returns a number. If x is positive, rounds towards $+\infty$; if x is negative, rounds towards $-\infty$. The default of n is 2. With complex x, rounds both the real and imaginary parts.

See also: **entier**, **int**, **round**, **xdf**.

`modf (x)`

Returns two numbers, the integral part of the number x and its fractional part. The integral part is rounded towards zero. Both the integral and fractional part of the return have the same sign as x. The sum of the two values returned equals x. The function actually is a wrapper to C's modf. With complex x, returns the integral and fractional parts for both its real and the imaginary part.

See also: \, **%**, **frac**, **entier**, **int**, **symmod**, **mod** assignment statement.

`muladd (x, y, z)`

The operator computes $x*y + z$ with extended internal precision, with x, y, z all numbers. See also: **fma**.

`multiple (x, y [, option])`

Checks whether numeric or complex x is a multiple of numeric y, i.e. whether x/y evaluates to an integral, and returns **true** or **false**.

Also returns **true** with $x = 0$ and any non-zero y.

If y is zero, **undefined** or $+/-$**infinity**, the function returns **fail**.

With complex x, returns **true** if both **real**$(x)/y$ and **imag**$(x)/y$ evaluate to the same integral, or if **real**$(x)/y$ evaluates to an integral and **imag**(x) is zero.

By passing the optional third argument **true**, a tolerant check is done, with subnormal x or y first converted to zero, and a subsequent approximate equality check to the nearest integer of x/y. The tolerance value internally used is the value of **DoubleEps** at the time of the function call.

In most cases, it may suffice to just call **integral**(x/y), which is twice as fast as this function.

`nan (x)`

Checks whether the number or complex number x evaluates to **undefined** (NaN). The operator returns **true** or **false**.

See also: **finite**, **float**, **math.isordered**.

x <u>nand</u> y

The operator returns the bitwise complement Boolean `and`, a signed integer: ~~(x **&&** y).

See also: **bytes.nand32**.

x <u>nor</u> y

The operator returns the bitwise complement Boolean `or`, a signed integer: ~~(x **||** y).

See also: **bytes.nor32**.

nonzero (x)

Checks whether the number or complex number x is neither 0 nor 0+0*I, respectively. The operator returns **true** or **false**.

See also: **zero**.

odd (x)

Checks whether the number x is odd. The operator returns **true** if x is odd, and **false** otherwise. With non-integral numbers, the operator returns **false**. With the complex value x, the operator returns **fail**.

See also: **even**.

polar (z)

Transforms the complex number z in Cartesian notation or the number z to polar form and returns two numbers: the magnitude (modulus) and the argument (phase angle), in this order. If z is a number and is zero, or if z is complex and its real and imaginary parts equal zero, the function returns zero twice.

See also: **abs**, **argument**, **cabs**, **cis**.

proot (x, n)

Returns the principal n-th root of the number or complex value x. n must be a positive integer. The principal n-th root in the complex domain is the first root found starting from the positive real axis going counter-clockwise.

See also: **cbrt**, **hypot**, **hypot2**, **hypot3**, **hypot4**, **root**, **sqrt**.

`pytha (a, b)`

Computes the Pythagorean equation $c^2 = a^2 + b^2$, without undue underflow or overflow and treating subnormal numbers accordingly, for numbers `a, b`.

See also: **hypot**, **squareadd**.

`qmdev (o)`

The operator computes the sum of the squared deviations of each observation o_i in the sequence, register, or table `o`, from its arithmetic mean μ, i.e.

$$\sum_{i=1}^{n} (o_i - \mu)^2$$

The return should be divided either by the number of elements n in the distribution `o` to calculate its population variance, or by n - 1 to compute its sample variance.

The Knuth-Welford algorithm used by the operator tries to prevent round-off errors as much as possible.

See also: **stats.sd**, **stats.var**.

`recip (x)`

Returns the inverse $1/x$ of a number or complex number `x`.

See also: /, **fastmath.reciprocal**.

`root (x [, n])`

Returns the non-principal `n`-th root of the number or complex value `x`. `n` must be an integer and is 2 by default. Note, that since the function computes the non-principal root, with complex `x`, **root**$(x, n) \neq x$ ^ $(1/n)$. In the complex domain, the function returns the `n`-th root of `x` whose argument is nearest to the argument of `x`.
See also: **argument**, **cbrt**, **hypot**, **hypot2**, **hypot3**, **hypot4**, **proot**, **sqrt**.

`round (x [, d])`

Rounds the number `x` to its `d`-th digit, using the round-half-up method. The return is a number. If `d` is omitted, the number is rounded to the nearest integer. If `d` is positive, the function rounds to the `d`-th fractional digit. If `d` is negative, it rounds to the `d`-th integral digit. **round** treats positive and negative values symmetrically, and is therefore free of sign bias. With complex numbers `x`=a+I*b returns **round**(**real**(a), d) + I***round**(**imag**(a), d).

The following Agena code explains the algorithm used:

```
round := proc(x, d) is
   d := d or 0;  # assign zero if d is null
   return int((10^d)*x + sign(x)*0.5) * (10^(-d))
end;
```

See also: **ceil**, **entier**, **int**, **mdf**, **xdf**, **math.rint**.

scalbn (x, n)

Just an alias for **ldexp**.

sec (x)

Returns the secant $\frac{1}{\cos(x)}$ as a number (in radians). The function is implemented in Agena and included in the lib/library.agn file. The function works on both numbers and complex values.

sech (x)

Returns the hyperbolic secant as a number (in radians). The function is implemented in Agena and included in the lib/library.agn file. The function works on both numbers and complex values.

sign (x)

Determines the sign of the number or complex value x. The result of the operator is determined as follows:

- 1, if real(x) > 0 or real(x) = 0 and imag(x) > 0
- -1, if real(x) < 0 or real(x) = 0 and imag(x) < 0
- 0 otherwise, even for -0.

If x is **undefined**, **sign** returns **undefined**.

See also: **math.copysign**, **math.flipsign**, **math.mulsign**, **signum**, **|** operator.

signum (x)

Determines the sign of the number or complex value x. If x is a number, the result of the operator is determined as follows:

- 1, if $x \geq 0$
- -1 otherwise.

With complex x, the operator returns $x/|x|$.

If x is **undefined**, **signum** returns **undefined**.

See also: **math.copysign**, **math.mulsign**, **sign**, **|** operator.

`sin (x)`

The operator returns the sine of x (in radians). Complex numbers are supported.

See also: **math.sincos**, **math.sinpi**.

`sinc (x)`

The operator returns the un-normalised cardinal sine of x (in radians), i.e. $\sin(x)/x$, with sinc(0) = 1. Complex numbers are supported.

See also: **cosc**, **math.rectangular**, **tanc.**

`sinh (x)`

The operator returns the hyperbolic sine of x (in radians). Complex numbers are supported. See also: **cosh**, **tanh**, **math.sinhcosh**.

`sqrt (x)`

Returns the square root of x.

If x is a number and negative, the operator returns **undefined**.

With complex numbers, the operator returns the complex square root, in the range of the right halfplane including the imaginary axis.

See also: **cbrt**, **hypot**, **hypot2**, **hypot3**, **invsqrt**, **proot**, **root**, **fastmath.sqroot**, **fastmath.sqrtfast**.

`square (x)`

The operator squares the number or complex number x and returns $x**2$.

See also: ******, **^**, **cube** operators, **math.issquare**, **squareadd**.

`tan (x)`

The operator returns the tangent of x (in radians). Complex numbers are supported. See also: **math.tanpi**.

`tanc (x)`

The operator returns the un-normalised cardinal tangent of x (in radians), i.e. $\tan(x)/x$, with tanc(0) = 1. Complex numbers are supported.

See also: **cosc**, **math.rectangular**, **sinc.**

`tanh (x)`

The operator returns the hyperbolic tangent of x (in radians). Complex numbers are supported.

See also: **cosh**, **sinh**.

`xdf (x [, n])`

Rounds down the number x at its n-th decimal place towards zero and returns a number. This is equivalent to truncating a float at its n-th decimal place. The default of n is 2.

With complex x, rounds both the real and imaginary parts.

See also: **entier**, **int**, **round**, **mdf**.

`x xnor y`

With numbers, the operator returns the bitwise complement Boolean `xor`, a signed integer: ~~(x ^^ y). With Booleans, returns **not**(x **xor** y), sometimes also called `if-and-only-if` (iff). See also: **bytes.xnor32**.

`x xor y`

With Booleans, returns x <> y. With non-booleans: returns the first operand if the second operand evaluates to **null**, otherwise the second operand will be returned.

See also: **^^**, **bytes.xor32**.

`zero (x)`

Checks whether the number or complex number x is 0 or 0+0*I, respectively. The operator returns **true** or **false**.

See also: **nonzero**.

11.1.2 math Library

This library is an interface to the standard C math library. It provides all miscellaneous functions inside the table `math`.

`math.accu ([init [, method]])`

Returns a factory that gets a number with each call, adds it to an internal accumulator, and returns the accumulated sum. If the factory is called with no argument, the current accumulated sum will be returned.

The function can be used if high accuracy numeric addition is needed. The initial value of the accumulator is 0. If `init`, a number, is given, the accumulator is set to `init` instead.

The function automatically takes care of storing and processing internal correction values - so the user does not have to worry about this.

By default, Neumaier summations is used. By passing a `method` (of type string), you may use an alternative algorithm to add numbers:

method	algorithm
`'babuska'`	Kahan-Babuška summation, highest accuracy but slowest
`'kahan'`	classic Kahan summation, lowest accuracy but fastest
`'kbn'`	Kahan-Babuška-Neumaier compensated summation, used in the Julia programming language
`'neumaier'`	Neumaier summation, good accuracy and performance (default)
`'ozawa'`	Kahan-Ozawa summation
`'raw'`	no auto-correction

See also: **factory.count**, **math.koadd**.

`math.arccosh (x)`

Returns the inverse hyperbolic cosine of the number `x` and returns a number. It works in the real domain only, and is a simple port of the acosh C function.

See also: **arccosh**.

`math.arcsinh (x)`

Returns the inverse hyperbolic sine of the number `x` and returns a number. It works in the real domain only, and is a simple port of the asinh C function.

See also: **arcsinh**.

```
math.arctanh (x)
```

Returns the inverse hyperbolic tangent of the number x and returns a number. It works in the real domain only, and is a simple port of the atanh C function.

See also: **arctanh**.

```
math.branch (x [, d [, subs]])
```

Returns its argument x - a number - if x is non-negative, otherwise returns 0. By passing any non-negative optional number d (the `direction`), the return is the same.

By passing any non-positive optional number d, returns x if it is negative, otherwise returns 0.

If x should be **undefined**, you can return any other number by passing the optional argument $subs$, which is **undefined** by default.

See also: **math.clip**, **math.wrap**, end of Chapter 11.1.2 for a comparison chart.

```
math.ceillog2 (x)
```

Returns the smallest exponent to 2 equals or greater than x, i.e. **ilog2(**x - 1) + 1, where x is a positive integer. If $x=$ 1, the result is 0; if $x <$ 1, **undefined** will be returned.

See also: **ilog2**, **math.ceilpow2**.

```
math.ceilpow2 (x)
```

Finds the smallest power of 2 greater than or equal to x, where x is a non-negative integer. If $x = 0$, the result is 1; if $x < 0$, **undefined** will be returned. Examples: math.ceilpow2(3) $\Rightarrow 4 = 2^2$, and math.ceilpow2(8) $\Rightarrow 8 = 2^3$.

The function returns **fail** if $x \geq 2^{31}$.

See also: **ilog2**, **math.ceillog2**, **math.floorpow2**, **math.ispow2**.

```
math.chop (x [, eps [, method [, n]]]])
```

Shrinks a number or complex number x more or less near zero to exactly zero, using one of many methods, passed as an integer. The default for eps is **Eps**. The standard $method$ is 0 for hard shrinking. n is used in the SmoothGarrote $method$.

method	Comment	Value	Domain
0	"Hard", performs hard shrinking	0 x	$\lvert x \rvert \leq$ eps $\lvert x \rvert >$ eps
1	"Soft", performs soft shrinking	0 sign(x) ($\lvert x \rvert$ - eps)	$\lvert x \rvert \leq$ eps $\lvert x \rvert >$ eps
3	"PiecewiseGarrote"	0 x - eps^2/x	$\lvert x \rvert \leq$ eps $\lvert x \rvert >$ eps
4	"SmoothGarrote"; with $n \rightarrow \infty$, goes to "Hard" shrinking	$x^{2n+1}/(x^{2n} + \text{eps}^{2n})$	any x
5	"Hyperbola"	0 sign(x) $\sqrt{x^2 - \text{eps}^2}$	$\lvert x \rvert \leq$ eps $\lvert x \rvert >$ eps

Method 2 has not been implemented. The function is a port of Mathematica's Chop function.

See also: **math.clip**, **math.unitise**, **math.zeroin**.

math.cld (x, y)

Returns the largest integer less than or equal to the real quotient $\frac{x}{y}$ of the numbers x and y.

See also: \ operator, **math.fld**.

math.clip (x [, a [, b [, f]]])
math.clip (x, a)

In the first form, returns x clipped to be between a and b. The return is x if $a \leq x \leq b$, a if $x < a$, and b if $x > b$. By default $a = -1$, $b = +1$. If function f is given which should return one numeric result, then if x is not in [a, b], the result of f(x) will be returned.

In the second form, returns x clipped to be between -a and +a, -a if $x < a$ and +a if $x > +a$.

See also: **calc.sigmoid**, **heaviside**, **math.branch**, **math.chop**, **math.rectangular**, **math.unitise**, **math.wrap**, end of Chapter 11.1.2 for a comparison chart.

math.compose (coeffs [, b])

Takes a table, sequence or register of coefficients coeffs and a base b and returns the composed number. In coeffs, the highest-order digit as the first element and the lowest-order digit as the last element. By default, the base is 10. The function does not take care of potential overflows. It is the complement to **math.decompose**.

`math.congruentprime (n [, a [, b]])`

Determines whether integer `n` is a prime number congruent to `a` modulo `b` - or in other words: a prime of the form `bn + a`, and returns it; otherwise for `n`, returns the next prime number congruent to `a` modulo `b`. By default, `a` is 3 and `b` is 4.

The function is implemented in Agena and included in the lib/library.agn file.

See also: **math.nextprime**.

`math.convertbase (s, a, b)`

Converts a number `s` or a number represented as a string `s` from base `a` to base `b`. `a` and `b` must be integers in the range 1 to 36. The number in `s` must be an integer of any sign. Floats are not allowed. The return is a string. The function is implemented in Agena and included in the lib/library.agn file.

See also: **math.decompose**, **math.ndigits**.

`math.copysign (x, y)`

Returns a number with the magnitude of `x` and the sign of `y`, i.e. abs(x) * sign(y). If `y` is 0, then its sign is considered to be 1. It is a plain binding to C's copysign function and does not post-process its result.

See also: **math.flipsign**, **math.mulsign**, **math.signbit**, **sign**, **signum**.

`math.cospi (x)`

Returns $\cos(\pi * x)$ for number `x` with better precision than calling the respective standard operator.

See also: **cos**, **math.sincos**, **math.sincospi**, **math.sinpi**, **math.tanpi**.

`math.dd (x)`

Converts a number `x` representing a sexagesimal number in TI-30 DMS format into its decimal representation, and returns a number. For example: 10.3045 representing 10°30'45" returns 10.5125.

The function is implemented in Agena and included in the lib/library.agn file.

See also: **math.dms**, **math.splitdms**, **math.todecimal**, **math.tosgesim**.

`math.decompose (x [, b])`

Splits an integer `x` to the base `b` into its digits and returns them in a sequence, with the highest-order digit as the first element and the lowest-order digit as the last

element. Any sign of x is ignored. By default, the base is 10, but you may choose any other positive base.

Example:

```
> b := 256;

> math.decompose(15 * b^2 + 7 * b + 1, 256):
seq(15, 7, 1)
```

See also: **math.compose**, **math.convertbase**, **math.ndigits**.

math.dirac (x [, eps])

The Dirac delta function, also known as the impulse function, returns 0 for all numbers x other than 0, and **infinity** if $x = 0$, iff eps is set to zero which is the default.

If eps is set to any positive value x, returns $1/(2*eps)*exp(-|x|/eps)$ even if $x = 0$.

math.dms (x)

Converts a number representing a decimal number x into its TI-30 sexagesimal DMS representation and returns a number. For example: 10.5125 returns 10.3045, representing 10°30'45".

See also: **math.dd**, **math.splitdms**, **math.todecimal**, **math.tosgesim**.

math.eps ([x [, option]])

The function returns the machine epsilon, the relative spacing between the number $|x|$ and its next larger number in the machine's floating point system. If no argument is given, x is set to 1.

On x86 machines and with Agena numbers, i.e. C doubles, eps(1) and eps() return $2.2204460492503e{-}016 = 2^{52}$, and eps(2) returns $4.4408920985006e{-}016 = 2^{51}$.

When given any second argument, the function computes a `mathematical` epsilon value that is also dependent on the magnitude of its argument x. It can be used in difference quotients, etc., for it prevents huge precision errors with computations on very small or very large numbers. The mathematical epsilon with respect to x is equal to $x*$ **sqrt(math.eps**(x)).

See also: **math.epsilon**, **math.nextafter**.

math.epsilon (x [, method])
math.epsilon (f, x [, ···] [, iters=n])

In the first form, by default returns the relative spacing between $|x|$ and its next larger number on the machine's floating point system, taking into account the magnitude of its argument. In this case, the function works like **math.eps** with the

true option set, but is 20 percent faster. If |x| < 1, you may choose a constant epsilon value yourself, e.g. Eps.

In the first form, you may choose between different `method`s to determine an epsilon value, where ulp = **math.nextafter**(x, **infinity**) - x:

Method	Formula
0 (default)	x * **sqrt**(ulp)
1	x * **cbrt**(ulp)
2	**sqrt**(ulp) * (x + **sqrt**(ulp))
3	**cbrt**(ulp) * (x + **cbrt**(ulp))

In the second form, by passing a function f and an argument x, the function determines an epsilon value by taking the function value $f(x)$ into account, using a divided difference table. If f is multivariate, pass its further arguments tight after x.

`math.expminusone (x)`

Returns a value equivalent to exp(x) - 1, with x a number. It is computed in a way that is accurate even if x is near 0, since exp(~0) and 1 are nearly equal.

The function can be used, for example, in financial mathematics, to calculate small daily interest rates, among other things.

See also: **expx2**, **math.lnplusone**.

`math.exponent (x)`

Returns the exponent e of a number x such that **math.mantissa**(x) * 2^e equals x. The result is identical to the second result returned by **frexp**. The function is around 20 percent faster but returns correct results only if your system supports IEEE 754 floating-point numbers, whereas **frexp** always works regardless of the internal representation.

See also: **frexp**, **math.mantissa**, **math.uexponent**.

`math.fall (x, n)`

The falling factorial function computes x*(x - 1)*(x - 2)* ... *(x - n + 1), with x a number and n an integer. If n is negative, the rising factorial function (Pochhammer function) is computed.

See also: **fact**, **math.pochhammer**.

`math.fdim (x, y [, a])`

The function returns x - y if its argument x, a number, is greater than or equal y, else it returns a, which is 0 by default.

`math.fib (n)`

Returns the n-th Fibonacci number, with n a non-negative integer. If $n > 76$, the function returns **fail** since the result is too large to be accurately represented. The defaults are: **math.fib**(0) = 0 and **math.fib**(1) = 1; with all other values computed by **math.fib**(n) := **math.fib**(n - 2) + **math.fib**(n - 1).

See also: **math.fibinv**.

`math.fibinv (n)`

For any non-negative integer n returns the index i of the Fibonacci number with **fib**(i) $<=$ n $<$ **fib**(i + 1). The function is implemented in Agena and included in the lib/library.agn file.

See also: **math.fib**.

`math.fld (x, y)`

Returns the largest integer less than or equal to the real quotient $\frac{x}{y}$ of the numbers x and y.

See also: \ operator, **math.cld**.

`math.flipsign (x, y)`

Returns the number x with its sign flipped if y (a number) is negative. For example, abs(x) = flipsign(x, x).

See also: **math.copysign**, **math.signbit**, **sign**.

`math.floorpow2 (x)`

Finds the largest power of 2 less than or equal to x, where x is a non-negative integer. If $x < 2$, the result is x. If $x < 0$, **undefined** will be returned. Examples: math.floorpow2(3) $\Rightarrow$ 2 = 2^1, and math.floorpow2(8) $\Rightarrow$ 8 = 2^3.

The function returns **fail** if $x \geq 2^{31}$.

See also: **ilog2**, **math.ceillog2**, **math.ceilpow2**, **math.ispow2**.

`math.fpclassify (x)`

For the given number x, returns

- 0 if x is **undefined** (= constant **math.fp_nan**),
- 1 if x is infinite, i.e. +/-**infinity** (= constant **math.fp_infinite**),
- 2 if x is subnormal (= constant **math.fp_subnormal**,
- 3 if x is zero (= constant **math.fp_zero**),
- 4 if x is normal (= constant **math.fp_normal**), including irregular values $\geq 2^{52}$.

Thus, for example, `ordinary` numbers are represented by results greater than 2.

The function returns **fail** if it could not determine the type of floating-point number (of C type double). It is a platform-independent port of C's fpclassify.

See also: **math.isnormal**, **math.issubnormal**.

math.fraction (x [, err])

Given a number x, this function outputs two integers and a number: the numerator n, the denominator d, and the accuracy epsilon, such that x := n / d to the accuracy epsilon := | (x - n/d) / x | $\leq$ err.

The error err should be a non-negative number, and by default is 0.

The function is implemented in Agena and included in the lib/library.agn file.

See also: **div** package.

math.frexp (x [, option])

Returns the sign s, the mantissa m and the exponent e of the number x, in this order, such that $s*m*2^e = x$. The sign is -1 if x is negative (including -0) and 1 otherwise. The mantissa is a float in the range [0.5, 1) except for $x = 0$, where the result is 0. The exponent is a negative or positive integer or zero.

If any option is given, then instead of the sign the sign bit s will be returned: 1 if x is negative or -0, and 0 otherwise. In this case $x = signum(-s) * m*2^e$.

The function works correctly only on IEEE 754-compliant systems.

See also: **frexp**, **ldexp**, **ilog2**, **math.exponent**, **math.mantissa**.

math.gcd (x, y)

Returns the greatest common divisor of the numbers x and y as a number. If x or y is not an integral, 1 will be returned. The function is implemented in Agena and included in the lib/library.agn file.

See also: **math.lcm**.

`math.hEps`

Deprecated. Represented the value 1.4901161193847656e-12, an epsilon value more or less in the middle of **Eps** and **DoubleEps**. See also: **hEps** constant, **environ.kernel/hEps**.

`math.iscube (n)`

Checks if a given integer `n` is a perfect cube, i.e. if **cbrt**(n)$^\wedge 3 = n$.

See also: **cbrt**, **cube**, **math.issquare**.

`math.isfib (n)`

Checks whether the non-negative integer `n` is a Fibonacci number.

See also: **math.fib**, **math.fibinv**.

`math.isinfinity (x)`

Returns -1 if its numeric argument `x` is **-infinity**, $+1$ if `x` is $+$**infinity**, or 0 if neither.

See also: **finite**, **infinite**.

`math.isirregular (x)`

Checks whether a number or complex number can be represented exactly on your system. I returns:

- **false** if $|x| < 2^{52}$: a number with decimal places can internally be represented as a number with decimal places, but not necessarily itself. With n < 52, the spacing between two subsequent representable numbers is the *fraction* 2^{n-52}.
- **fail** if $2^{52} \le |x| \le 2^{53} + 1$: representable numbers are exactly the integers; spacing between representable numbers is exactly 1.
- **true** if $|x| > 2^{53} + 1$: an integer mostly cannot be exactly represented; with n > 52, spacing is the *integer* 2^{n-52}.

With complex `x`, checks whether at least the real or imaginary part evaluates to **false** or **fail** - according to the rule mentioned above - and returns that; otherwise if both parts evaluate to **true**, returns **true**.

`math.isminuszero (x)`

Returns **true** if `x` is -0 (minus zero) and **false** otherwise. See also: **math.signbit**.

`math.isnormal (x)`

Returns **true** returns true if a number is neither +0, -0, +infinity, -infinity, undefined nor subnormal. The result is equal to the expression **math.fpclassify**(x) = **math.fp_normal**.

With complex x, returns **math.isnormal(real**(x)) **and math.isnormal(imag**(x)).

See also: **finite**, **math.isminuszero**, **math.issubnormal**.

`math.isordered (x, y)`

Returns **false** if at least one of its arguments x and y - two numbers - is **undefined**, and **true** otherwise. See also: **nan**.

`math.ispow2 (x)`

Checks whether a given non-negative number x is a power of base 2 ($x = 2^{log2(x)}$) and returns **true** or **false**. Also returns **false** if x is negative.

The function returns **fail** if its argument is ±**infinity** or **undefined**.

`math.isprime (x)`

Returns **true**, if the integral number x is a prime number, and **false** otherwise. Note that you have to take care yourself that x is an integer and is less than the largest integer representable on your system.

See also: **math.nextprime**, **math.prevprime**.

`math.isqrt (x)`

Returns the integer square root of the number x: the largest integer m such that m*m ≤ x.

`math.issquare (n)`

Checks if a given integer n is a perfect square, i.e. if sqrt(n)^2 = n. Any power of two is a perfect square, for example n = 1, 4, 9, 16, 25, 36, 49, etc.

See also: **sqrt**, **square**, **cube**, **math.iscube**, **math.ispow2**.

`math.issubnormal (x)`

Checks whether the number x is subnormal, i.e. whether internally the leading digit of its mantissa is 0. The function returns **true** or **false**. Subnormal numbers are very close to zero, have reduced precision and lead to excessive CPU usage. They are in the range [-2.2250738585072009e-308, -4.9406564584124654e-324] and [4.9406564584124654e-324, 2.2250738585072009e-308]. 0, **undefined** and

+/-**infinity** are not subnormal. Please note that the next representable number after 0 (towards $+\infty$) is subnormal.

With complex x, returns **math.issubnormal(real(x)) or math.issubnormal(imag(x))**.

See also: **math.normalise**, **math.smallestnormal**, **math.two54**, **math.zerosubnormal**.

```
math.koadd (x, y [, q])
```

The function adds x and y using Kahan-Ozawa round-off error prevention and returns two numbers: the sum of x and y plus the updated value of the correction variable q. The optional correction variable q should be 0 at first invocation, and the previously returned correction variable otherwise - if q is not given, it defaults to 0.

The following algorithm used is:

```
math.koadd := proc(s :: number, x :: number, q) is
   local sold, u, v, w, t;
   q := optnumber(q, 0);
   v := x - q;
   sold := s;
   s := s + v;
   if abs(x) < abs(q) then
      x, q := -q, x
   fi;
   u := (v - x) + q;
   if abs(sold) < abs(v) then
      sold, v := v, sold
   fi;
   w := (s - sold) - v;
   q := u + w;
   return s, q
end;
```

A typical usage should look like:

```
x, q -> 0;
y := 0.1;
while x < 1 do
   x, q := math.koadd(x, y, q)
od;
print(s + q);
```

See also: **math.accu**, **stats.sumdata**.

```
math.largest
```

This constant represents the largest positive number representable in Agena. It is computed during start-up and may be different from the setting returned by **environ.system**, the latter statically compiled into the Agena binary. The smallest negative number (nearest to $-\infty$) is the negative of this constant, i.e. - **math.largest**.

See also: **math.lastcontint**, **math.smallest**.

`math.lastcontint`

This constant represents the largest integer i on the floating-point system such that i - 1 <> i. In other words: The constant represents the largest integer value that can be stored in an Agena number without loss of precision. On 32-bit systems (and higher), it is equal to $2^{53} = 9,007,199,254,740,992$.

See also: **math.largest**.

`math.lcm (x, y)`

Returns the least common multiple of to numbers `x` and `y` as a number. The function is implemented in Agena and included in the lib/library.agn file.

See also: **math.gcd**.

`math.lnabs (x)`

Returns **ln**(**abs**(x)) for numeric or complex `x`. With complex numbers, takes care of underflows.

`math.lnplusone (x)`

Returns a value equivalent to ln(1 + `x`), with `x` a number. It is computed in a way that is accurate even if `x` is near zero.

It can be used, for example, in financial calculations, when computing small daily interest rates.

Example: ln(1.0000000000000001) ⇒0, math.lnplus1(0.0000000000000001) ⇒ 1e-016.

See also: **math.expminusone**.

`math.logs (x, b)`

The iterated logarithm of `x`, log*(x) (for `log star`) returns the number of times the logarithm function to a given base `b` must be iteratively applied on `x` until the result reaches or drops below 1. If `x` <= 1, returns 0. The algorithm is equivalent to:

```
> logs := proc(x, b) is
>    for i from 0 while x > 1 do
>        x := log(x, b)
>    od;
>    return i
> end;
```

`math.mantissa (x)`

Returns the mantissa m of a number x such that m * 2^**math.exponent**(x) equals x. The result is identical to the first result returned by **frexp**, and is in the range [0.5, 1) (or zero when x is zero). The function is around 20 percent faster but returns correct results only if your system supports IEEE 754 floating-point numbers, whereas **frexp** always works regardless of the internal representation.

See also: **frexp**, **math.exponent**, **math.significand**.

`math.max (x [, ···])`

Returns the maximum value among its arguments of type number.

`math.min (x [, ···])`

Returns the minimum value among its arguments of type number.

`math.modinv (a [, m])`

Computes the modular (multiplicative) inverse a^{-1} of an integer a (modulo m) such that $a * a^{-1} = 1$ (modulo m). m is 257 by default. If m is not prime, then not every non-zero integer a has a modular inverse - in this case the function returns **undefined**.

See also: **%** operator, **drem**, **iqr**, **symmod**, **math.modulus**.

`math.modulus (x, y)`

The function is a plain binding to the C `%` modulus operator. Both its arguments must be integers. The return is an integer. If $y = 0$, the function returns **undefined**.

See also: **%** operator, **bytes.mod32**, **drem**, **hashes.fibmod32**, **iqr**, **symmod**, **math.modinv**, **math.nearmod**.

`math.morton (x, y)`

Interleaves the bits of integers x and y, so that all of the bits of x are in the even positions and y in the odd; the function can be used to linearising 2D integer co-ordinates, combining x and y into a single integer that can be compared easily. It has the property that a number is usually close to another if their x and y values are close.

`math.mulsign (x, y)`

Multiplies, not copies, its first argument with the sign of its second, and returns x * **signum**(y).

See also: **math.copysign**, **math.flipsign**, **math.signbit**, **sign**, **signum**.

`math.ndigits (x [, b])`

Returns the number of integer digits - without decimal places - in the number x to the base b. By default, b is 10.

If b is -10, counts the number of decimal places (fractional digits) in x, where x is considered to be of base 10. This feature is experimental and not failsafe.

See also: **math.decompose**, **math.nthdigit**.

`math.nearbyint (x)`

Returns x rounded to the nearest integer, returns the same result as **round**(x, 0) does but is implemented differently and 5 % faster. The function has been included for C math library compatibility reasons only.

See also: **int**, **math.rint**, **math.trunc**.

`math.nearmod (x, m)`

Returns the closest value to the given number x divisible by the given modulus m, equivalent to round(x/m) * m. See also: **%**, **math.modulus**.

`math.nextafter (x, y)`

Returns the next machine floating-point number of x in the direction toward y.

See also: **+++** and **---** operators, **math.eps**, **math.ulp**.

`math.nextmultiple (n, b)`

Returns the next multiple of an integer n to the given base b, towards +**infinity** if b is positive, and towards -**infinity**, if b is negative.

`math.nextpower (x, base [, option])`

By default returns the smallest power of `base` greater than x. If the third argument is **true**, then the smallest power of `base` greater than *or* equal to x will be returned.

`math.nextprime (x)`

Returns the smallest prime greater than the given number x.

See also: **math.congruentprime**, **math.prevprime**, **math.isprime**.

`math.norm (x, a1:a2 [, b1:b2])`

Converts the number x in the scale [$a1$, $a2$] to one in the scale [$b1$, $b2$]. The second and third arguments must be pairs of numbers. If the third argument is missing, then x is converted to a number in [0, 1]. The return is a number.

See also: **linalg.scale**, **math.wrap**, **stats.scale**.

`math.normalise (x [, option])`

Checks whether its numeric argument x is subnormal and in this case normalises it, i.e. returns a non-zero normalised value $x*2^{64}$ that is close to x; otherwise returns its argument x unaltered. If any `option` is given, the unsigned high 4-byte word of the result will be returned, too.

With complex x, normalises both its real and imaginary part if necessary and returns the complex number **math.normalise**(real(x)) + **I*math.normalise**(imag(x)); the `option` is not supported in this case.

It is useful to prevent excessive CPU usage with values very close to zero.

For more information, see **math.issubnormal**.

See also: **math.zeroin**, **math.zerosubnormal**.

`math.nthdigit (x, n)`

Returns the n-th digit of the number x, with n an integer. To evaluate an integer digit, n should be positive; for a decimal place, n should be negative.

The function is written in Agena and included in the lib/library.agn file.

See also: **math.ndigits**.

`math.piecewise (cond₁, f₁, cond₂, f₂, ..., condₙ, fₙ [, fotherwise])`

Evaluates a piecewise-continuous function. $cond_1$, etc. are relations evaluating to Booleans, and f_1, etc. numeric expressions. The arguments are checked from left to right and as soon as a condition $cond_k$ is met, **piecewise** returns the respective value f_k. If no condition meets, the function returns `fotherwise`, and **undefined** if not given.

The implementation is far from perfect as *all* of its arguments are evaluated before executing the procedure. Better use the Boolean operator **and** and **or**, for example:

- `math.piecewise(x < 2, -1, x < 3, 1, infinity)` and
- `x < 2 and -1 or x < 3 and 1 or infinity`

are equivalent, but the latter is around 15 times faster due to application of the McCarthy Rule.

`math.pochhammer (x, n)`

Computes the Pochhammer function (rising factorial), where both x and n are real numbers. It returns the number:

$$\frac{\Gamma(x+n)}{\Gamma(x)}$$

See also: **fact**, **math.fall**.

`math.powmod (x, p, m)`

Computes $x \wedge p$ % m.

`math.prevprime (x)`

Returns the largest prime less than the given number x.

See also: **math.nextprime**, **math.isprime**.

`math.quadrant (x)`

This function returns the quadrant of an angle x given in radians and returns an integer in [1, 4].

`math.ramp (x)`

For number x, gives x if $x > 0$ and 0 otherwise.

See also: **heaviside**, **math.rectangular**.

`math.random ([m [, n]])`

When called without arguments, returns a pseudo-random float with uniform distribution in the range [0,1).

When called with two integers m and n, **math.random** returns a pseudo-random integer with uniform distribution in the range [m, n].

The call `math.random($n$)`, for a positive n, is equivalent to `math.random(1, $n$)`. The call `math.random(0)` produces an integer with all bits (pseudo) random.

This function uses the xoshiro256** algorithm to produce pseudo-random 64-bit integers, which are the results of calls with argument 0. Other results (ranges and floats) are unbiased extracted from these integers.

Agena initializes its pseudo-random generator with the equivalent of a call to **math.randomseed** with no arguments, so that **math.random** should generate different sequences of results each time the program runs.

`math.randoms ([m [, n]] [, option])`

This function creates random numbers as Agena did before version 2.27.10.

When called without arguments, returns a pseudo-random real number in the range (0,1). It can generate up to 2 * **environ.maxlong** unique random numbers in this interval.

When called with a number `m`, **math.random** returns a pseudo-random integer in the range [1, `m`].

When called with two numbers `m` and `n`, **math.random** returns a pseudo-random integer in the range [`m`, `n`].

If `option`, any Boolean, is given, then the sequence of values returned should be arbitrary, otherwise it is always the same unless **math.randomseed** is called with other values.

See also: **math.randomseed**, **skycrane.dice**.

`math.randomseed ([x, y])`

When called with at least one argument, the integer parameters `x` and `y` are joined into a 128-bit seed that is used to reinitialize the pseudo-random generator; equal seeds produce equal sequences of numbers. The default for `y` is zero.

When called with no arguments, Lua generates a seed with a weak attempt for randomness.

This function returns the two seed components that were effectively used, so that setting them again repeats the sequence.

To ensure a required level of randomness to the initial state (or contrarily, to have a deterministic sequence, for instance when debugging a program), you should call **math.randomseed** with explicit arguments.

`math.randomseeds ([x, y])`

Sets `x` and `y` as the `seeds` for the pseudo-random generator, as Agena did before version 2.27.10: equal seeds produce equal sequences of numbers. `x` and `y` must both be positive integers. It returns two new settings. The function does not check for `x`= 0x464fffff and `y` = 0x9068ffff.

If called without arguments, the function returns the current seeds.

See also: **math.random**.

`math.rectangular (x [, pi])`

`math.rectangular (x [, a [, b [, pi]]])`

In the first form, computes the rectangular pulse function for number `x`:

$$math.rectangular(x) = \begin{cases} 1 & \text{if } |x| < 0.5 \\ 0.5 & \text{if } |x| = 0.5 \\ 0 & \text{if } |x| > 0.5 \end{cases}$$

In the second form, `a` represents the rising edge, and `b` the falling edge of the rectangular pulse function. By default, `a` = -0.5 and `b` = +0.5. The function then returns 0 if `x` < `a` or `x` > `b`; 0.5 if (`x` = `a` or `x` = `b`) and `a` <> `b`, and 1 otherwise.
If `pi` is the Boolean value **true**, the function computes the box distribution Pi(x):

$$Pi(x) = \begin{cases} 1 & \text{if } |x| < 0.5 \\ \text{undefined} & \text{if } |x| = 0.5 \\ 0 & \text{if } |x| > 0.5 \end{cases}$$

See also: **heaviside**, **math.clip**, **math.ramp**, **math.triangular**, **math.unitise**, **sinc**.

`math.relerror (a, b)`

Computes the relative error $|b - a| / |a|$, handling case of **undefined** and **infinity**.

`math.rempio2 (x [, option])`

Conducts an argument reduction of `x` into the range $|y| < \frac{\pi}{2}$ and returns y = `x` - N*$\frac{\pi}{2}$. If any `option` is given, then the function also returns N, or actually the last three digits of N. The number of operations conducted are independent of the exponent of the input.

The function is 60 percent faster than **math.wrap**, but returns a result different from `x` if its argument $|x|$ is already in the range $\frac{\pi}{4}$ .. $\frac{\pi}{2}$.

This function is just a port to the underlying C function rem_pio2 which is used to compute sines, cosines and tangents.

See end of Chapter 11.1.2 for a comparison chart.

`math.rint (x)`

Rounds a (complex) float to a (complex) integer according to the current rounding method which you can query and set with **environ.kernel/rounding**.

See also: **ceil**, **entier**, **int**, **mdf**, **round**, **math.nearbyint**.

`math.signbit (x)`

Checks whether the number x has its sign bit set and returns **true** or **false**. It is a plain binding to C's copysign function. For example, although -0 = 0, **math.signbit**(-0) $\Rightarrow$ **true** and **math.signbit**(0) $\Rightarrow$ **false**.

See also: **math.copysign**, **math.flipsign**, **math.isminuszero**, **sign**.

`math.significand (x)`

Returns the mantissa of number x in a normalised form, in the range[1, 2), with **math.significand**(x) = 2***math.mantissa**(x) = **ldexp**(x, -**ilog2**(x)). If x is 0, the return is 0.

See also: **math.uexponent**.

`math.sincos (x)`

Returns both the sine and cosine for number or complex x as two numbers or complex numbers. The function is around 10 to 15 % faster than calling the **sin** and **cos** operators separately.

See also: **cos**, **sin**, **math.sincosfast**, **math.sincospi**.

`math.sinhcosh (x)`

For number x, returns both the hyperbolic sine and hyperbolic cosine as two numbers. The function is around 30 to 35 % faster than calling the **sinh** and **cosh** operators separately.

With complex x, returns complex results.

See also: **cosh**, **sinh**.

`math.sincospi (x [, option])`

Returns both $\sin(\pi*x)$ and $\cos(\pi*x)$ for number x with better precision than calling the respective standard operators. If option is **true**, than the tangent, i.e. $\tan(\pi*x)$ is returned too.

See also: **math.sincos**, **math.sinpi**, **math.cospi**, **math.tanpi**.

`math.sinpi (x)`

Returns $\sin(\pi*x)$ for number x with better precision than calling the respective standard operator.

See also: **sin**, **math.sincos**, **math.sincospi**, **math.cospi**, **math.tanpi**.

`math.smallest`

This constant represents the smallest positive number representable in Agena. It is computed during start-up and is different from the setting returned by **environ.system**, the latter statically compiled into the Agena binary.

See also: **math.largest**.

`math.smallestnormal`

This constant denotes the smallest positive normal number representable on your system.

`math.splitdms (x)`

Splits the number x representing a sexagesimal number in TI-30 DMS format into its parts and returns three numbers: the degrees, minutes, and seconds. For example: -10.3045 represents -10°30'45".

The function is implemented in Agena and included in the lib/library.agn file.

See also: **math.dd**, **math.dms**, **math.todecimal**, **math.tosgesim**.

`math.stirnum (n, k, kind)`

Computes either the Stirling number for n, k, of the first or the second kind. `kind` is 1 by default to compute the first kind; you can set it to 2 to evaluate the second kind.

`math.tanpi (x)`

Returns $\tan(\pi*x)$ for number x with better precision than calling the respective standard operator.

See also: **tan**, **math.sincos**, **math.sincospi**, **math.sinpi**, **math.cospi**.

`math.todecimal (h [, m [, s]])`

Converts a sexagesimal time value given in hours h, minutes m and seconds s into its decimal representation. The optional arguments m and s default to 0. If a

sexagesimal value is negative, then `h` should be negative, while `m` and `s` should be non-negative.

Example:

```
> math.todecimal(12, 30, 1):  # half past noon and one second
12.500277777778
```

See also: **clock.todec**, **math.todms**, **math.tosgesim**.

`math.todms (x)`

Converts a number in DMS notation to its decimal representation, e.g. 10.3045, representing 10°30'45", returns 10.5125.

See also: **math.todecimal**.

`math.tohex (x)`

Converts a non-negative integer `x` in the range [0, 255] to its hexadecimal representation, returned as a 2-character string.

See also: **math.convertbase**.

`math.toradians (d [, m [, s]])`

Returns the angle given in degrees `d`, minutes `m` and seconds `s`, in radians. The optional arguments `m` and `s` default to 0.

`math.tosgesim (d)`

Converts a decimal time value given by the number `d` into its sexagesimal representation and returns three numbers: the hours, minutes, and seconds.

Example:

```
> math.tosgesim(12.500277777778):
12      30      1
```

The function is written in Agena and included in the lib/library.agn file.

See also: **math.dms**, **math.todecimal**, **math.todms**.

`math.triangular (x)`
`math.triangular (x [, a [, b]])`

In the first form, computes the triangular function of base length 1 for number `x`:

$$\text{math.triangular(x)} = \begin{cases} 1 - |2x| & \text{if } |x| < 0.5 \\ 0 & \text{if } |x| \geq 0.5 \end{cases}$$

In the second form, by passing a left and a right border `a`, `b`, the function returns non-zero values in this range, and 0 otherwise, with `a` = -0.5 and `b` = +0.5 the defaults. Thus, the general formula used by the function is:

math.triangular(`x`, `a`, `b`) := max(0, 1 - |2*(`x` - offset)/d|),

where d := |`b` - `a`| and offset := `a` + d/2.

See also: **heaviside**, **math.branch**, **math.clip**, **math.rectangular**, **math.unitise**, **math.wrap**, **sinc**.

`math.trunc (x)`

Returns `x` rounded to the nearest integer towards zero, returns the same result as **int**(`x`). The function has been included for C math library compatibility reasons only.

See also: **math.nearbyint**.

`math.two54`

The constant represents 2^{54}, a value with which subnormal numbers can be multiplied in order to become normal. See also: **math.issubnormal**.

`math.uexponent (x [, option])`

Computes the unbiased base-2 exponent of number `x`, i.e. returns math.exponent(`x`) - 1, except for `x` = 0 and subnormal numbers where the result is -1023, and for `x` = **undefined** or `x` = ±**infinity** returns 1024.

If any `option` is given, then returns **sign**(`x`)***math.uexponent**(`x`), but for `x` = **undefined** returns 0x401= 1025, for `x` = -**infinity** returns -1024, and for `x` = **infinity** returns +1024. Due to the definition, returns 0 for `x` = 0 and subnormal `x`.

See also: **bytes.getunbiased**, **math.significand**, **frexp**.

`math.ulp (x [, eps])`

Computes the unit of least precision (ULP), the spacing between floating-point numbers, for number `x`, as a measure of accuracy in numeric calculations. It is equivalent to **math.nextafter**(`x`, **infinity**) - `x`.

If `eps` is given, the function also returns the number of ULPs - an integer - between `x` and `x` + `eps`.

`math.unitise (x [, eps])`

Returns 0 if its number argument x is zero or close to zero, and 1 otherwise:

- 0 if $|x| \leq$ `eps`,
- 1 if $|x| >$ `eps`.

With complex numbers $x = a + I*b$, returns

- 0 if $|a| \leq$ `eps` and $|b| \leq$ `eps`,
- 1 if $|a| >$ `eps` and $|b| >$ `eps`.

By default, `eps` is set to the constant Eps.

See also: **heaviside**, **math.clip**, **math.rectangular**, **math.unitstep**, **math.zeroin**.

`math.unitstep (x [, eps])`

For number x, gives 0 for $x < 0$ and 1 otherwise.

See also: **heaviside**, **math.unitise**.

`math.wrap (x [, a [, b]])`

Conducts a range reduction of the number x to the interval [a, b) and returns a number. If $x \in [a, b)$, x is simply returned.

In the second form, if a is not given, a is set to $-\pi$ and b to $+\pi$. If a is given but not b, a is set to $-a$ and b to $+a$, so a should be positive.

The result is equivalent to:

```
> dec x, a;
> dec b, a;
> a + (b + x symmod b) symmod b:
```

See also: **%** operator, **math.branch**, **math.clip**, **math.norm**, **math.rempio2**, **zx.reduce**, end of Chapter 11.1.2 for a comparison chart.

`math.xlnplusone (x)`

Computes x - ln(1 + x) in a way that is accurate even if x is near zero. The algorithm is ten percent faster than simply returning x - **math.lnplusone**(x).

`math.zeroin (x)`

Sets a number or complex number x to 0 if $|x| \leq$ **DoubleEps**. With a complex number x, return 0+I*0 if its magnitude $|x| \leq$ **DoubleEps** or sets its respective parts to zero if their respective absolute values are less or equal to **DoubleEps**.

See also: **math.chop**, **math.normalise**, **math.zerosubnormal**.

```
math.zerosubnormal (x)
```

Checks whether its numeric argument x is subnormal and in this case returns 0, otherwise returns its argument x. It is useful to prevent excessive CPU usage in case of arguments very close to zero. Note that result retains the sign of x.

With complex x, returns the complex number **math.zerosubnormal**(**real**(x)) + **I*math.zerosubnormal**(**imag**(x)).

For more information, see **math.issubnormal**, **math.normalise**, **math.zeroin**.

Comparison of some clipping functions:

```
> gdi.plotfn([<< x -> math.rempio2(x) >>,
>    << x -> math.wrap(x) >>,
>    << x -> math.clip(x) >>,
>    << x -> zx.reduce(x) >> ,
>    << x -> math.branch(x) >>],
>    -4, 4, -4, 4,
>    colour=['red', 'navy', 'green', 'black', 'maroon']);
```
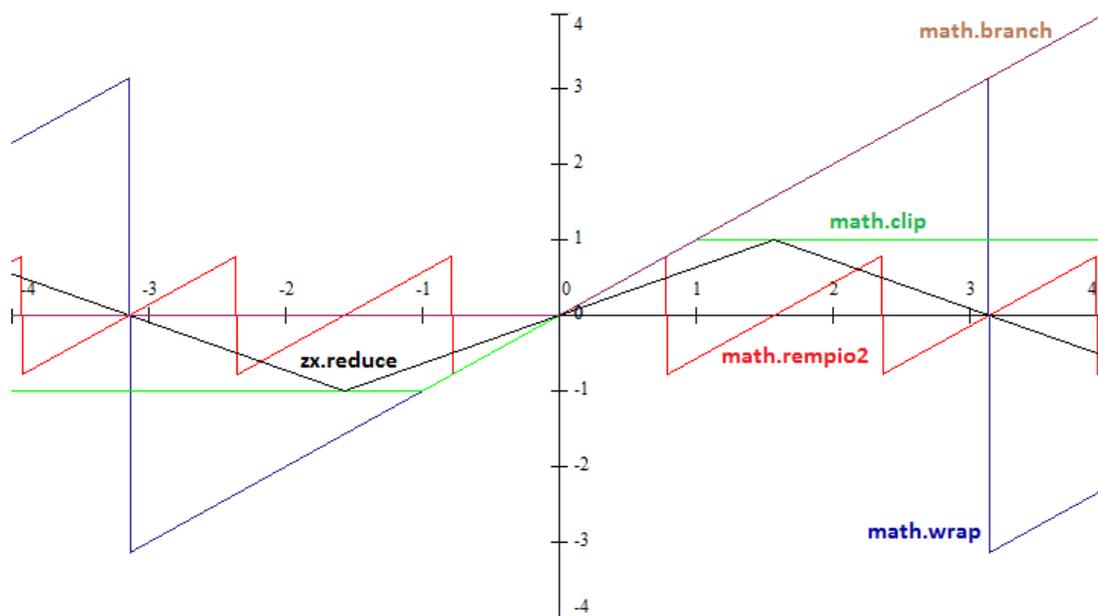

11.1.3 fastmath Library

As a *plus* package, this library is not part of the standard distribution and must be activated with the **import** statement, e.g. `import fastmath`.

The library provides procedures to approximate mathematical functions in the real domain. Despite its name, the package functions may not necessarily be faster than the standard functions and operators implemented in Agena.

`fastmath.cosfast (x)`

Approximates **cos**(x) for number x, and returns a number. It is around 40 percent faster than **cos**.

See also: **cos**.

`fastmath.hypotfast (x, y)`

Returns the hypotenuse of the two numbers x and y; the return is a number. The function is sixty percent faster than **hypot**, but prone two round-off errors.

`fastmath.invroot (x [, degree [, n [, xhalf]]])`

Approximates the inverse root 1/root(x, $degree$) using the Quake III method, and returns a number. x is the radicand, $degree$ the degree-th root which by default is 2. n is the number of iterations to be conducted and by default is $2\hat{~}degree$. $xhalf$ is the internal equivalent of x, 0.5*x by default. The greater the $degree$, the less accurate is the result.

See also: **fastmath.reciprocal**, **fastmath.sqroot**.

`fastmath.invsqrt (x)`

Approximates the inverse square root 1/**sqrt**(x), using Quake's Fast Inverse Square Root method and returns a number. It is five percent faster than the inverse of the **sqrt** operator.

`fastmath.lbfast (x)`

Approximates **log2**(x) for number x, and returns a number. It is around a third faster than **log2**. If $x <= 0$, the result will be wrong.

See also: **log2**.

`fastmath.reciprocal (x)`

Approximates the reciprocal of its argument x of type number. The return is a number. The function is purely experimental.

See also: **fastmath.invroot**, **fastmath.sqroot**.

`fastmath.sinfast (x)`

Approximates **sin**(x) for number x, and returns a number. It is around 40 percent faster than **sin**.

`fastmath.sincosfast (x)`

Returns both an approximation of the sine and cosine as two numbers. The function is around 10 % faster than calling **math.sincos**.

`fastmath.sqroot (x)`

Roughly approximates the square root of its argument x of type number. The returns are two numbers: guesses computed using C doubles and floats, in this order. The function is purely experimental.

See also: **fastmath.invroot**, **fastmath.reciprocal**, **fastmath.sqrtfast**.

`fastmath.sqrtfast (x)`

Approximates the square root of its argument x of type number. The function is purely experimental.

See also: **fastmath.sqroot**.

`fastmath.tanfast (x)`

Approximates **tan**(x) for number x, and returns a number. It is around 40 percent faster than **tan**.

See also: **tan**.

11.2 bytes Library

As a *plus* package, this library is not part of the standard distribution and must be activated with the **import** statement, e.g. `import bytes`.

The library provides procedures for bit and byte twiddling.

11.2.1 General Functions

`bytes.bcd (n)`

Returns the Binary coded decimal (BCD) representation of the non-negative integer `n`. From left to right, each decimal digit is converted to a four-bit representation (0 = 0b0000, 9 = 0b1001), and the resulting bit sequence is then returned as one decimal integer, e.g. decimal 102 = 0001 0000 0010 ⇒ BCD 258.

By default, if only `n` is given, the function converts the decimal integer to BCD. If **true** is passed as a second argument, `n` is converted from BCD to its decimal integer representation.

`bytes.castint (x, bits)`

Casts number `x` to a C integer. The results may be platform-dependent.

bits	Cast to
8	uint8_t
16	uint16_t
32	uint32_t
64	uint64_t
-8	int8_t
-16	int16_t
-32	int32_t
-64	int64_t

`bytes.fpbtoint (x)`

Converts a `floating point byte` generated by **bytes.inttofpb** back. This function is used to evaluate numbers transported to the Lua/Agena virtual machine. Please note that **math.inttofpb(math.fpbtoint(**x**))** does not return x.

`bytes.numhigh (x)`

Returns the higher bytes of a number x as an integer. The function does not support complex numbers.

See also: **bytes.numwords**, **bytes.numlow**.

`bytes.numlow (x)`

Returns the lower bytes of a number x as an integer. The function does not support complex numbers.

See also: **bytes.numhigh**, **bytes.numwords**, **bytes.setlow**.

`bytes.numwords (x)`

Returns both **bytes.numhigh**(x), **bytes.numlow**(x) plus the unbiased exponent (i.e. **math.exponent**(**bytes_numhigh**(x)) - 1, except for x = 0 -> -1023) as three results, in this order.

See also: **bytes.setnumwords**.

`bytes.inttofpb (x)`

Converts the integer x to a `floating point byte`, represented as (eeeeexxx), where the real value is (1xxx) * 2^(eeeee - 1) if eeeee <> 0 and (xxx) otherwise. This function is used to transport numbers to the Lua/Agena virtual machine.

See also: **bytes.fpbtoint**.

`bytes.leadzeros (x)`

Returns the number of leading zeros in the unsigned 32-bit integer x, and the modified value of x after this operation, where all bits starting with the first non-zero bit in x are set to 1.

See also: **bytes.leastsigbit**, **bytes.mostsigbit**, **bytes.onebits**, **bytes.trailzeros**.

`bytes.leastsigbit (x)`

Returns the position of the least significant bit (lsb) in the unsigned 32-bit integer x, here the smallest index of the first 1-bit, counting from bit index 1. If x < 1, returns 0.

See also: **bytes.leadzeros**, **bytes.mostsigbit**, **bytes.onebits**.

`bytes.mostsigbit (x)`

Returns the position of the most significant bit (msb) in the unsigned 32-bit integer `x`, i.e. the largest index of a 1-bit, counting from bit index 1. If `x` < 1, returns 0.

See also: **bytes.leadzeros**, **bytes.leastsigbit**, **bytes.mostsigbit**.

`bytes.onebits (x)`

Returns the number of bits set in the unsigned 32-bit integer `x`.

See also: **bytes.leadzeros**, **bytes.mostsigbit**.

`bytes.optsize (n)`

For a given number of bytes `n`, calculates the optimal number of bytes (places) in a C `array` (e.g. a memfile, numarray or even a string) if it shall be aligned on the 4- or 8-byte word boundary.

See also: **strings.strlen**.

`bytes.pack (fmt, v1, v2, ···)`

Returns a binary string containing the values `v1`, `v2`, etc. packed (that is, serialised in binary form) according to the format string `fmt`.

The first argument to **bytes.pack**, **bytes.packsize**, and **bytes.unpack** is a format string, which describes the layout of the structure being created or read.

A format string is a sequence of conversion options. The conversion options are as follows:

> `<`: sets little endian
> `>`: sets big endian
> `=`: sets native endian
> `![n]`: sets maximum alignment to `n` (default is native alignment)
> `b`: a signed byte (char)
> `B`: an unsigned byte (char)
> `h`: a signed short (native size)
> `H`: an unsigned short (native size)
> `l`: a signed long (native size)
> `L`: an unsigned long (native size)
> `j`: a lua_Integer
> `J`: a lua_Unsigned
> `T`: a size_t (native size)
> `i[n]`: a signed int with `n` bytes (default is native size)
> `I[n]`: an unsigned int with `n` bytes (default is native size)

`f`: a float (native size)

`d`: a double (native size)

`n`: a lua_Number

`cn`: a fixed-sized string with `n` bytes

`z`: a zero-terminated string

`s[n]`: a string preceded by its length coded as an unsigned integer with `n` bytes (default is a size_t)

`x`: one byte of padding

Xop: an empty item that aligns according to option op (which is otherwise ignored)

`' '`: (empty space) ignored

(A "`[n]`" means an optional integral numeral.) Except for padding, spaces, and configurations (options "xX <=>!"), each option corresponds to an argument (in **bytes.pack**) or a result (in **bytes.unpack**).

For options "!n", "sn", "in", and "ln", n can be any integer between 1 and 16. All integral options check overflows; **bytes.pack** checks whether the given value fits in the given size; **bytes.unpack** checks whether the read value fits in a Lua integer.

Any format string starts as if prefixed by "!1=", that is, with maximum alignment of 1 (no alignment) and native endianness.

Alignment works as follows: For each option, the format gets extra padding until the data starts at an offset that is a multiple of the minimum between the option size and the maximum alignment; this minimum must be a power of 2. Options "c" and "z" are not aligned; option "s" follows the alignment of its starting integer.

All padding is filled with zeros by **bytes.pack** (and ignored by **bytes.unpack**).

See also: **bytes.packsize**, **bytes.unpack**, **math.ispow2**.

bytes.packsize (fmt)

Returns the size of a string resulting from **bytes.pack** with the given format. The format string cannot have the variable-length options 's' or 'z'.

bytes.reverse (x)

Reverses all the bits in the unsigned 32-bit integer `x`, flipping all bits from 0 to 1 and vice versa.

bytes.setnumhigh (x, i)

The function sets the higher bytes of the number `x` to the unsigned 32-bit integer `i`, and returns the new number. It does not support complex numbers.

See also: **bytes.setnumlow**, **bytes.numhigh**.

`bytes.setnumlow (x, i)`

The function sets the lower bytes of the number `x` to the unsigned 32-bit integer `i`, and returns the new number. It does not support complex numbers.

See also: **bytes.setnumhigh**, **bytes.numlow**.

`bytes.setnumwords (hx, lx)`

Returns the number (C double) x represented by the unsigned 32-bit integers `hx` and `lx`, i.e. x = **bytes.setnumwords(math.numhigh**(x)**, math.numlow**(x)**)**.

See also: **bytes.numwords**.

`bytes.tobig (x [, order])`

On Little Endian systems, converts the number `x` into its Big Endian representation and returns it. On Big Endian platforms, just returns `x` unaltered. If `order` is 4, then the function processes `x` as an unsigned 4-byte integer. If `order` is -4, the function treats `x` as a signed 4-byte integer.

See also: **bytes.tolittle**, **os.endian**.

`bytes.tobinary (x)`

Converts a non-negative integer into its binary representation, a sequence of zeros and ones.

See also: **math.convertbase**.

`bytes.tobytes (x [, nbytes [, false]])`

If given no option, returns a sequence of eight bytes representing the number `x` in Little Endian order, i.e. the least-significant byte is the first entry in the resulting sequence. If `nbytes` is the number +4 or -4, `x` is a assumed to be an unsigned 4-byte integer or signed 4-byte integer, respectively, and a sequence of four bytes representing `x` in Little Endian representation will be returned. If `nybtes` is +2, x will be treated as an unsigned 2-byte integer, with a sequence of two bytes to be returned.

On Big Endian systems, conversion to Little Endian representation can be switched off by passing a third argument, the Boolean value **false**.

See also: **getbit**, **getbits**, **getnbits**, **bytes.tonumber**.

`bytes.tolittle (x [, order])`

On Big Endian systems, converts the number `x` into its Little Endian representation and returns it. On Little Endian platforms, just returns `x` unaltered. If `order` is 4, then the function processes `x` as an unsigned 4-byte integer. If `order` is -4, the function treats `x` as a signed 4-byte integer.

See also: **bytes.tobig**, **os.endian**.

`bytes.tonumber (s)`

Takes a sequence `s` of two, four or eight numbers representing bytes and converts it into an Agena number. Regardless of your platform, the order of bytes in `s` is assumed to be Little Endian.

If `s` contains eight bytes, it is assumed to represent a C unsigned double. If it contains four bytes, an unsigned four-byte integer is assumed; and with two bytes, an unsigned two-byte integer is assumed.

See also: **bytes.tobytes**.

`bytes.trailzeros (x)`

Returns the number of trailing zeros in the unsigned 32-bit integer `x`, and the modified value of `x` after this operation, where all bits starting with the least significant bit in `x` are set to 1.

See also: **bytes.leadzeros**, **bytes.leastsigbit**, **bytes.mostsigbit**, **bytes.onebits**.

`bytes.unpack (fmt, s [, pos])`

Returns the values packed in string `s` (see **bytes.pack**) according to the format string `fmt`. An optional `pos` marks where to start reading in `s` (default is 1). After the read values, this function also returns the index of the first unread byte in `s`.

11.2.2 cast Functions

The bytes package provides the 'cast' userdata data structure representing an Agena number as both a C double (i.a. Agena number) and its two higher and lower 32-bit unsigned integer representations, along with functions to query and assign its individual components.

Example:

```
> import bytes
```

```
> a := bytes.cast(-Pi):
cast(-3.1415926535898 : 3221823995, 1413754136)

> hx := bytes.gethigh(a):
3221823995

> hx >>> 31:  # sign bit (1 = minus, 0 = plus)
1

> bytes.sethigh(a, hx && 0x7fffffff):  # absolute value
3.1415926535898
```

bytes.cast ([x])

bytes.cast ([hx, lx])

Creates a userdata structure of type 'cast' that stores the Agena number 0 or x and its integer representation as two unsigned 32-bit integers. Technically, the userdata represents the C union (see source file src/sunpro.h):

Big-Endian platforms	Little-Endian platforms
```typedef union { double value; struct { uint32_t msw; uint32_t lsw; } parts; } ieee_double_shape_type;```	```typedef union { double value; struct { uint32_t lsw; uint32_t msw; } parts; } ieee_double_shape_type;```

If no argument is given, then the userdata represents zero (0), alternatively you can set it to number x (first form).

You may also initialise the userdata by passing both its unsigned 32-bit integer word hx and unsigned 32-bit integer lower word lx (second form).

See also: **bytes.getdouble**, **bytes.gethigh**, **bytes.getlow**, **bytes.getwords**, **bytes.setdouble**, **bytes.setwords**.

bytes.getdouble (a)

Returns the floating point element of 'cast' userdata a, i.e. a number.

bytes.gethigh (a)

Returns the higher unsigned 32-bit integer representation of a number from 'cast' userdata a.

See also: **bytes.getlow**, **bytes.getwords**.

`bytes.getlow (a)`

Returns the lower unsigned 32-bit integer representation of a number from 'cast' userdata `a`.

See also: **bytes.gethigh**, **bytes.getwords**.

`bytes.getunbiased (a)`

Returns the unbiased exponent of the double x represented by 'cast' userdata `a`. Equals **math.exponent**(x) - 1, except for x = 0 where the result is -1023. If |x| < 1, the result is always negative.

See also: **bytes.gethigh**, **bytes.getlow**, **bytes.getwords**, **math.uexponent**.

`bytes.getwords (a)`

Returns both the higher and lower unsigned 32-bit integer representations of a number from 'cast' userdata `a`.

See also: **bytes.getdouble**, **bytes.gethigh**, **bytes.getlow**.

`bytes.setdouble (a, x)`

Sets the floating point element of 'cast' userdata `a` and returns the higher and lower unsigned 32-bit integer representations, in this order.

See also: **bytes.setwords**.

`bytes.sethigh (a, hx)`

Sets the higher unsigned 32-bit integer element `hx` of 'cast' userdata `a`. The return is the corresponding floating point representation, i.e. a number.

See also: **bytes.setdouble**, **bytes.setlow**.

`bytes.setlow (a, lx)`

Sets the lower unsigned 32-bit integer element `lx` of 'cast' userdata `a`. The return is the corresponding floating point representation, i.e. a number.

See also: **bytes.setdouble**, **bytes.sethigh**.

bytes.setwords (a, hx, lx)

Sets the higher and lower unsigned 32-bit integer elements `hx` and `lx` of 'cast' userdata `a`. The return is the corresponding floating point representation, i.e. a number.

See also: **bytes.setdouble**, **bytes.sethigh**, **bytes.setlow**.

11.2.3 IEEE754 Functions

The bytes package provides the 'ieee' userdata data structure representing an Agena number as both a C double (i.a. Agena number) and its components sign bit, biased exponent and high- and low-word mantissa. See **bytes.ieee** for details.

Example:

```
> import bytes

> exp10 := proc(x) is
>    x *:= log2(10);
>    local i := round(x);
>    local f := bytes.ieee(0);
>    bytes.setieee(f, expo = i + 1023);
>    x -:= i;
>    return bytes.getieee(f, 'double') *
>        (1.0 + x*(0.69314718055994530941723212145818 +
>              x*(0.24022650695910071233355126316333 +
>              x*(0.05550410866482157995314226376862  +
>              x*(0.00961812910762847716197907157365899 +
>              x*(0.00133335581464284434234122219877996 +
>              x*(0.00015403530393381609954437097332742 +
>              x*(0.00001525273380405984028002543901201 +
>              x*(0.00000132154867901443094884037582282888 +
>              x*0.00000010178086009239699727490007597745)))))))))
>end;
```


bytes.ieee ([x])

bytes.ieee ([signbit, exponent, high_mantissa, low_mantissa])

Creates a userdata structure of type 'ieee' that stores the Agena number 0 or `x` and allows read and write access to its components sign bit `signbit`, its biased `exponent`, the high-word part of the mantissa `high_mantissa` and its low-word part `high_mantissa`. Technically, the userdata represents the C union (see source file src/sunpro.h):

Big-Endian platforms	Little-Endian platforms

``
typedef union {
  double v;
  struct {
    uint64_t sign   : 1;
    uint64_t exponent   : 11;
    uint64_t mantissa_high : 20;
    uint64_t mantissa_low : 32;
  } c;
} double_ieee754;
``
	``
	typedef union {
	  double v;
	  struct {
	    uint64_t mantissa_low : 32;
	    uint64_t mantissa_high : 20;
	    uint64_t exponent : 11;
	    uint64_t sign : 1;
	  } c;
	} double_ieee754;
	``

If no argument is given, then the userdata represents zero (0), alternatively you can set it to number `x` (first form).

You may also initialise the userdata by passing both its sign bit, exponent, and the high and low parts of the mantissa (second form).

bytes.setieee (a, options)

Sets components in 'ieee' structure `a`. Accepted `options` are one or more pairs:

- `'double'`: any Agena number,
- `'signbit'`: the sign bit, 1 for minus, 0 for plus,
- `'expo'`: the biased exponent, an unsigned 4-byte integer,
- `'high'`: high-word part of the mantissa,
- `'low'`: low-word part of the mantissa.

Example:

```
> bytes.setieee(a, double=Pi, signbit=1);  # sets −Pi to 'ieee' object a
```


bytes.getieee (a [, options])

Returns the components of 'ieee' structure `a`. If only `a` is given, then its five components number, sign bit, exponent, mantissa high-word and mantissa low-word are returned, in this order.

If one or more of the following strings are passed as `options`, then the requested components will be returned in the order given by the user:

- `'double'`: any Agena number,
- `'signbit'`: the sign bit, 1 for minus, 0 for plus,
- `'expo'`: the biased exponent, an unsigned 4-byte integer,
- `'high'`: high-word part of the mantissa,
- `'low'`: low-word part of the mantissa.

Example:

```
> bytes.getieee(a, 'double', 'signbit'):
−3.1415926535898          1
```

Following are specialised functions for 'ieee' data:

bytes.setieeesignbit (a, bit [, option])

Sets the sign bit of ieee structure `a` to `bit`, which is either 1 for minus or 0 for plus. If any `option` is given, the function returns the updated values for double, sign bit, exponent, high and low-word mantissa, in this order.

bytes.setieeeexpo (a, e [, option])

Sets the biased exponent of ieee structure `a` to non-negative integer `e`. If any `option` is given, the function returns the updated values for double, sign bit, exponent, high and low-word mantissa, in this order.

bytes.setieeehigh (a, hi [, option])

Sets the high-word of the mantissa of ieee structure `a` to non-negative integer `hi`. If any `option` is given, the function returns the updated values for double, sign bit, exponent, high and low-word mantissa, in this order.

bytes.setieeelow (a, lo [, option])

Sets the high-word of the mantissa of ieee structure `a` to non-negative integer `lo`. If any `option` is given, the function returns the updated values for double, sign bit, exponent, high and low-word mantissa, in this order.

bytes.setieeedouble (a, v [, option])

Sets the floating-point component in ieee structure `a` to Agena number `v`. If any `option` is given, the function returns the updated values of the sign bit, exponent, high and low-word mantissa in `a`, in this order.

bytes.getieeedouble (a)

Returns the floating-point component in ieee structure `a`, an Agena number.

11.2.4 32-bit Integer Operations

The following functions process 32-bit signed and unsigned integers.

Please note, that by default, Agena including the functions listed below, work in unsigned mode. You can switch to signed operations by issuing

```
> environ.kernel(signedbits = true);
```

on the command line or in a (library) file.

bytes.add32 (a, b [, ···])

Adds two or more numbers `a`, `b`, ... using 4-byte unsigned integer arithmetic. The return is an integer.

You can switch from unsigned to signed arithmetic by setting **environ.kernel**(signedbits = **true**), and from signed to unsigned arithmetic by **environ.kernel**(signedbits = **false**).

See also: **&+** operator.

bytes.sub32 (a, b [, ···])

Subtracts two or more numbers `a`, `b`, ... using 4-byte unsigned integer arithmetic. The return is an integer.

You can switch from unsigned to signed arithmetic by setting **environ.kernel**(signedbits = **true**), and from signed to unsigned arithmetic by **environ.kernel**(signedbits = **false**).

See also: **&-** operator.

bytes.mul32 (a, b [, ···])

Multiplies two or more numbers `a`, `b`, ... using 4-byte unsigned integer arithmetic. The return is an integer.

You can switch from unsigned to signed arithmetic by setting **environ.kernel**(signedbits = **true**), and from signed to unsigned arithmetic by **environ.kernel**(signedbits = **false**).

See also: **&*** operator.

bytes.muladd32 (a, b [, ···])

Multiplies two numbers `a`, `b`, and adds further numbers `c`, ... using 4-byte unsigned integer arithmetic. The return is the integer $a*b + c + ...$

You can switch from unsigned to signed arithmetic by setting **environ.kernel**(signedbits = **true**), and from signed to unsigned arithmetic by **environ.kernel**(signedbits = **false**).

See also: **bytes.add32**, **bytes.mul32**.

`bytes.div32 (a, b [, ···])`

Divides two or more numbers `a`, `b`, ... using 4-byte unsigned integer arithmetic. The return is an integer.

You can switch from unsigned to signed arithmetic by setting **environ.kernel**(signedbits = **true**), and from signed to unsigned arithmetic by **environ.kernel**(signedbits = **false**).

See also: **&/** operator.

`bytes.mod32 (a, b)`

Takes the modulus `a` % `b` (with % the C modulus operator, not Agena's **%**), using 4-byte unsigned integer arithmetic. The return is an integer.

You can switch from unsigned to signed arithmetic by setting **environ.kernel**(signedbits = **true**), and from signed to unsigned arithmetic by **environ.kernel**(signedbits = **false**).

See also: **math.modulus**.

`bytes.divmod32 (a, b)`

Returns the quotient and remainder of the 4-byte division `a`/`b`.

See also: **bytes.div32**, **bytes.mod32**.

`bytes.and32 (···)`

Conducts a binary AND operation on all the arguments (none, one or multiple signed or unsigned 32-bit integers) and returns an integer.

See also: **&&** operators.

`bytes.arshift32 (x, n)`

Returns the 32-bit signed or unsigned integer `x` shifted `n` bits to the right. The number `n` may be any representable integer. Negative displacements shift to the left.

This shift operation is what is called arithmetic shift. Vacant bits on the left are filled with copies of the higher bit of x, thus preserving the sign of x; vacant bits on the right are filled with zeros. In particular, displacements with absolute values higher than 31 result in zero or 0xFFFFFFFF (all original bits are shifted out).

See also: <<< and >>> operators, **bytes.shift32**.

bytes.extract32 (n, field [, width])

Returns the unsigned number formed by the bits field to $field + width$ - 1 from n. Bits are numbered from 0 (least significant) to 31 (most significant). All accessed bits must be in the range [0, 31].

The default for $width$ is 1.

Signed 32-bit integers n are not supported.

See also: **bytes.replace32**.

bytes.isint32 (n)

Checks whether the given number n is in the range of a signed or an unsigned 4-byte integer and returns **true** or **false**.

To check in which mode Agena is, check the **environ.kernel/signedbits** setting. It should usually be unsigned.

If you are in unsigned mode, the argument should be in the range 0 .. **environ.kernel().maxulong** = 0 .. 4'294'967'295.

If you are in signed mode, n should be in the range **environ.kernel().minlong** .. **environ.kernel().maxlong** = -2'147'483'647 .. 2'147'483'647.

Example:

```
> import bytes

> environ.kernel().signedbits:  # we are in unsigned mode (C uint32_t's).
false

> bytes.isint32(4'294'967'295):
true

> bytes.isint32(4'294'967'295 + 1):
false
```

`bytes.mask32 (n)`

Returns an integer with the first `n` bits set to one, e.g. `bytes.mask32(3)` → 7.

`bytes.nand32 (···)`

Conducts a binary complementary OR operation on all the arguments (none, one or multiple signed 32-bit integers) and returns an integer. There is no `unsigned` mode available, as the results would be of no use.

See also: **nand**.

`bytes.nextbit (mask)`

Gets and clears the next bit from the unsigned 4-byte `mask`, starting with the most significant bit. The function returns the modified value of `mask` and the respective bit position 0 .. 31.

`bytes.nor32 (···)`

Conducts a binary complementary OR operation on all the arguments (none, one or multiple signed 32-bit integers) and returns an integer. There is no `unsigned` mode available, as the results would be of no use.

See also: **nor**.

`bytes.not32 (x)`

Conducts a binary NOT operation on the signed or unsigned 32-bit integer `x` and returns an integer.

See also: ~~ operator.

`bytes.numto32 (x)`

Converts a number `x` to its signed or unsigned 4-byte integer representation. Note that very large values (positive or negative) might overflow, e.g. bytes.numto32($2^{32}+1$) ⇒1. The result may differ across platforms in overflow situations.

You can switch from unsigned to signed arithmetic by setting **environ.kernel**(signedbits = **true**), and from signed to unsigned arithmetic by **environ.kernel**(signedbits = **false**).

See also: **math.uexponent**.

bytes.or32 (···)

Conducts a binary OR operation on all the arguments (none, one or multiple signed or unsigned 32-bit integers) and returns an integer.

See also: || operator.

bytes.parity32 (x)

Determines the parity of the unsigned 4-byte integer x, i.e. the number of 1-bits in x modulo 2.

Returns 0 if x is of even parity, and 1 in case of odd parity.

See also: **hashes.parity**.

bytes.replace32 (n, v, field [, width])

Returns a copy of n, an unsigned 32-bit integer, with the bits `field` to `field + width - 1` replaced by the value v.

Signed 32-bit integers n are not supported.

See **bytes.extract32** for details about `field` and `width`.

bytes.rotate32 (x, n)

Rotates the bits in the 32-bit integer x n displacements to the right if $n >= 0$, or n places to the left if $n < 0$. The return is a 32-bit integer.

Internally the function uses unsigned 32-bit integers by default. You can change this to signed integers by calling **environ.kernel** with the `'signedbits'` option.

See also: <<<< and >>>> operators.

bytes.shift32 (x, n)

Shifts the bits in the 32-bit integer x n displacements to the left if $n < 0$, and to the right if $n > 0$.

Internally the function uses unsigned 32-bit integers by default. You can change this to signed integers by calling environ.kernel with the `'signedbits'` option.

See also: <<< and >>> operators, **bytes.arshift32**.

`bytes.xnor32 (···)`

Conducts a binary complementary exclusive-OR operation on all the arguments (none, one or multiple signed 32-bit integers) and returns an integer. There is no `unsigned` mode available, as the results would be of no use.

See also: **xnor**.

`bytes.xor32 (···)`

Conducts a binary exclusive-OR operation on all the arguments (none, one or multiple signed or unsigned 32-bit integers) and returns an integer.

See also: ^ ^ operator.

11.3 mapm - Arbitrary Precision Library

As a *plus* package, in Solaris, Linux, Mac OS X, and Windows, this library is not part of the standard distribution and must be activated with the **import** statement, e.g. `import mapm`.

In OS/2 and DOS, the package is built into the binary executable and does not need to be activated with **import**.

The package provides functions to conduct arbitrary precision mathematics with real numbers. It uses Mike's Arbitrary Precision Math Library, written by Michael C. Ring.

Standard operators like +, -, *, /, ^, %, <, =, >, and unary minus are supported.

All function names in this library begin with the letter x.

The package uses its own kind of numbers which are different from Agena numbers: use **mapm.xnumber** and **mapm.xtonumber** to convert between them.

By default, the precision is set to 17 digits, but you can change this any time with the **mapm.xdigits** function, see example below:

It is always advised to pass numbers *as strings* if possible. This is because Agena uses C doubles which are not 100 % precise.

```
> import mapm;

> mapm.xdigits(100);   # precision set to 100 digits

> a := mapm.xnumber('0.5');

> a*mapm.Pi:
1.57079632679489662

> b := mapm.xnumber(0.5):
0.50000000000000000

> b*mapm.Pi:
1.57079632679489662
```

You cannot directly compare MAPM numbers with Agena numbers:

```
> a − b=0:
false

> a − b = mapm.xnumber(0):
true
```

See also: The **long** package implementing 80-bit floating-point arithmetic, described in Chapter 11.15.

The mathematical functions are:

Function	Meaning	Function	Meaning
mapm.xabs	absolute value	mapm.xfactorial	factorial
mapm.xarccos	arc cosine	mapm.xidiv	integer division
mapm.xarccosh	inverse hyperbolic cosine	mapm.xln	natural logarithm
mapm.xadd	addition	mapm.xlog	logarithm of given base
mapm.xarcsin	inverse sine	mapm.xlog2	base-2 logarithm
mapm.xarcsinh	inverse hyperbolic sine	mapm.xlog10	common logarithm
mapm.xarctan	inverse tangent	mapm.xmul	multiplication
mapm.xarctan2(x, y)	4 quadrant inverse tangent	mapm.xpow	power
mapm.xarctanh	hyperbolic inverse tangent	mapm.xsign	sign
mapm.xcbrt	cubic root	mapm.xsin	sine
mapm.xcos	cosine	mapm.xsincos	sine and cosine
mapm.xcosh	hyperbolic cosine	mapm.xsinh	hyperbolic sine
mapm.xdiv	division	mapm.xsqrt	square root
mapm.xerf	error function	mapm.xsub	subtraction
mapm.xerfc	complementary error function	mapm.xtan	tangent
mapm.xhypot (x, y [, prec])	hypotenuse with optional precision	mapm.xtanh	hyperbolic tangent
mapm.xterm (c, x, n)	computes $c*x^n$	mapm.xfma (x, y, z)	fused multiply-add
mapm.xexp	exponential function		
mapm.xexp2	base-2 exponentation		
mapm.xexp10	base-10 exponentation		

Most of the **mapm** functions accept a second argument - a non-negative integer - giving the individual precision.

The package provides the following metamethods:

Operator	Name	Description
+	'__add'	addition
-	'__sub'	subtraction
*	'__mul'	multiplication
/	'__div'	division
%	'__mod'	modulus
^	'__pow'	power with any exponent
**	'__ipow'	power with integer exponent

Operator	Name	Description
-	`'__unm'`	unary minus
<	`'__lt'`	less-than
=	`'__eq'`	equals
<=	`'__le'`	less-or-equal
abs	`'__abs'`	absolute value
sign	`'__sign'`	sign
recip	`'__recip'`	reciprocal
sqrt	`'__sqrt'`	square root
square	`'__square'`	square ($x \char`^ 2$)
cube	`'__cube'`	cube ($x \char`^ 3$)
ln	`'__ln'`	natural logarithm
exp	`'__exp'`	exponential function
sin	`'__sin'`	sine
cos	`'__cos'`	cosine
tan	`'__tan'`	tangent
arcsin	`'__arcsin'`	arcus sine
arccos	`'__arccos'`	arcus cosine
arctan	`'__arctan'`	arcus tangent
sinh	`'__sinh'`	hyperbolic sine
cosh	`'__cosh'`	hyperbolic cosine
tanh	`'__tanh'`	hyperbolic tangent
n/a	`'__gc'`	garbage collection
n/a	`'__tostring'`	conversion to a string, e.g. for the pretty printer

Other functions are:

Function	Meaning	Function	Meaning
mapm.xceil	ceil function	**mapm. xexponent**	exponent
mapm.xfloor	floor function	**mapm.xinv**	reciprocal
mapm.xiseven	test for even number	**mapm.xisint**	check for an integral
mapm.xisodd	test for odd number	**mapm.xmod**	modulus
mapm.xround	rounds downwards to the nearest integer	**mapm.xneg**	negates a number
mapm. xcompare(x, y)	comparison, returns -1 if x < y, 0 if x = y, and 1 if x > y	**mapm.xnumber**	converts an Agena number or a string representing a number to an arbitrary precision number

Function	Meaning	Function	Meaning
mapm.xdigits	sets the number of digits used in all subsequent calculations. With no argument, returns the current setting. (default is 17)	**mapm. xtonumber**	converts an arbitrary precision number to an Agena number. A second optional argument n gives the precision for this specific number, with n > 0.
mapm. xdigitsin	significant digits	**mapm.xtostring**	converts an arbitrary precision number to a string
mapm.xchebyt (n, x)	n-th Chebyshev polynomial of the first kind at point x		

Available constants with precision of 1,000 digits are:

Constant	Value	Comment
mapm.Pi	π	
mapm.Pi2	2π	
mapm.PiO2	$\pi/2$	
mapm.PiO4	$\pi/4$	
mapm.PiO180	$\pi/180$	radians per degree
mapm.InvPi2	$1/(2\pi)$	
mapm.InvPiO4	$4/\pi$	
mapm.InvPiSqO4	$4//\pi^2$	
mapm.E	$E = \exp(1)$	
mapm.sqrt2	$\sqrt{2}$	
mapm.sqrt3	$\sqrt{3}$	
mapm.ln2	$\ln(2)$	
mapm.Invln2	$1/\ln(2)$	
mapm.Invsqrt2	$1/\sqrt{2}$	
mapm.Phi	$(1 + \sqrt{5})/2$	Golden Ratio
mapm.InvPhi	$1/((1 + \sqrt{5})/2)$	inverse Golden Ratio
mapm.InvPhiSq	$(1/((1 + \sqrt{5})/2))^2$	
mapm.InPhi	$\ln(1 + \sqrt{5})/2$	logarithm of Golden Ratio
mapm.InvInPhi	$1/\ln(1 + \sqrt{5})/2$	inverse
mapm.naught **mapm.nought**	0	zero
mapm.one	1	
mapm.two	2	
mapm.three	3	
mapm.four	4	
mapm.five	5	
mapm.six	6	
mapm.seven	7	

Constant	Value	Comment
mapm.eight	8	
mapm.nine	9	
mapm.ten	10	
mapm.eleven	11	
mapm.twelve	12	
mapm.fifty	50	
mapm.hundred	100	
mapm.thousand	1,000	
mapm.half	0.5	
mapm.quarter	0.25	
mapm.tenth	0.1	
mapm.fifth	0.2	
mapm.hundredth	0.01	
mapm.thousandth	0.001	

These constants have been defined in source file lib/mapm.agn.

11.4 mp - GNU Multiple Precision Arithmetic Library

As a *plus* package, the **mp** package is not part of the standard distribution and must be activated with the **import** statement, i.e. `import mp`.

The mp library is a binding to the GMP library providing multiple functions to conduct signed and unsigned integer arithmetic of arbitrary precision.

The package provides various metamethods for easy entry of calculations, too.

Signed and unsigned integers - `mpints` for short in this context - are represented by mp userdata objects which can be passed to the functions and operators described below.

OS/2, Solaris, Linux and UNIX users may have to install the original GMP 6.1 library separately in order for this binding to work. The package is not available for Mac OS X. In order for this binding to work on Intel CPUs, you may need at least a Sandybridge processor.

```
> import mp

> a, b := mp.uint(1), mp.uint(2)  # unsigned integers

> a + b:
mp(3)

> a, b := mp.sint(2), mp.sint(3)  # signed integers

> a * b:
mp(6)
```

11.4.1 Creation of Signed and Unsigned Integers

mp.uint (n)

mp.uint (numstr [, base])

Creates an unsigned integer object (mpz_t GMP userdata object) from an unsigned integer `n`, or a string `numstr` representing an unsigned integer.

If you pass a string you may indicate whether it is in decimal format by passing the optional second argument 10, and if its is in hexadecimal encoding, pass 16, which is the default.

See also: **mp.sint**, **mp.setstring**.

`mp.sint (n)`

`mp.sint (numstr [, base])`

Like **mp.uint**, but creates a signed integer mpz_t GMP userdata object.

11.4.2 Signed and Unsigned Integer Arithmetic

The following operators and functions can be applied both on signed and unsigned mpints:

`mp.add (a, b)`

Adds two mpints `a`, `b`, and returns a new mpint. Used by __add metamethod, i.e.: mp.add(`a`, `b`) = `a` + `b`.

`mp.subtract (a, b)`

Subtracts two mpints `a`, `b` and returns a new mpint. Used by __sub metamethod, i.e.: mp.subtract(`a`, `b`) = `a` - `b`.

`mp.multiply (a, b)`

Multiplies two mpints `a`, `b` and returns a new mpint. Used by __mul metamethod, i.e.: mp.multiply(`a`, `b`) = `a` * `b`.

`mp.divide (a, b)`

Divides two mpints `a`, `b` and returns a new mpint. Used by __div metamethod, i.e.: mp.divide(`a`, `b`) = `a` / `b`.

`mp.addmul (r, a, b)`

Multiplies two mpints `a`, `b`, adds the result to `r` and returns the updated value of `r`: i.e.: mp.addmul(`r`, `a`, `b`) <=> `r` +:= `a` * `b`.

`mp.submul (r, a, b)`

Multiplies two mpints `a`, `b`, subtracts the result from `r` and returns the updated value of `r`: i.e.: mp.submul(`r`, `a`, `b`) <=> `r` -:= `a` * `b`.

`mp.modulus (r, a, b)`

Computes the modulus of two mpints `a`, `b` and returns a new mpint. Used by __mod metamethod, i.e.: mp.subtract(`a`, `b`) = `a` % `b`.

`mp.neg (a)`

Returns -`a`, with `a` an mpint, as a new mpint. Used by `__unm` metamethod, i.e. mp.neg(`a`) $<=>$ -`a`.

`mp.mul2exp (a, b)`

Computes `a` * 2$^{\wedge}$`b`, with `a`, `b` mpints, and returns the result in a new mpint. The operation is equivalent to a left shift by `b` bits.

`mp.tdiv (a, b)`

Returns both quotient and remainder of `a` / `b`, with `a`, `b` units, both rounded towards zero.

`mp.tdivq (a, b)`

Divides two mpints `a`, `b` and returns the resulting quotient as a new mpint, rounded towards zero.

`mp.tdivr (a, b)`

Divides two mpints `a`, `b` and returns the resulting remainder as a new mpint, rounded towards zero.

`mp.powm (a, b, c)`

With three units, computes (`a` $^{\wedge}$ `b`) % `c` and returns the result as a new mpint.

`mp.root (a, n)`

Computes the truncated integer part of the `n`-th root of `a`, with `a` an mpint, `n` a number, and returns the result as a new mpint.

`mp.log2 (a)`

Like **ilog2**, but for mpint `a`.

11.4.3 Number Theoretic Functions

`mp.testprime (a [, reps])`

Checks whether mpint `a` is a prime and returns:

- 0 if `a` is no prime,
- 1 if `a` is probably prime,
- 2 if `a` is definitely prime.

The accuracy of the result can be controlled by the optional second parameter `reps` which by default is 15, with reasonable values between 15 and 50.

mp.nextprime (a)

Returns the next prime to mpint `a`, as a new mpint.

mp.gcd (a, b)

Returns the greatest common divisor of mpints `a` and `b`, as a new mpint.

mp.gcdext (s, t, a, b)

Returns the greatest common divisor of mpint `a` and mpint `b` and returns the result.

In addition the function sets mpints `s` and `t` to coefficients satisfying `a`*`s` + `b`*`t` = (the return).

mp.lcm (a, b)

Returns the least common multiple of mpints `a` and `b`, as a new mpint.

mp.invert (a, b)

Computes the inverse of mpint `a` modulo mpint `b` and returns a mpint.

mp.jacobi (a, b)

Computes the Jacobi symbol (`a`/`b`) of the mpints `a`, `b`, which is defined only if `b` is odd. The return is a new mpint.

mp.legendre (a, p)

Computes the Legendre symbol (`a`/`p`) of the mpints `a`, `b`, which is defined only if `p` is an odd positive prime. The return is a new mpint.

mp.kronecker (a, b)

Computes the Jacobi symbol (`a`/`b`) for mpints `a`, `b`, with the Kronecker extension (`a`/2)=(2/`a`) with odd `a`, and (`a`/2)=0 with even `a`. The return is a new mpint.

mp.remove (a, b)

Removes all occurrences of the factor `a` from `b` returns the result as a new mpint.

mp.factorial (n)

Returns the factorial of *number* `n` (_not_ mpint) as a new mpint.

`mp.fib (n)`

Returns the `n`-th Fibonacci number, with `n` a number (_not_ mpint), and returns a new mpint.

`mp.lucas (n)`

Sets the `n`-th Lucas number, with `n` a number (_not_ mpint), and returns a new mpint.

`mp.primorial (n)`

Returns the primorial of number `n` (_not_ mpint), i.e. the product of all positive prime numbers <= `n`. The return is a new mpint.

`mp.binomial (n, k)`

Computes the binomial coefficient `n` (an mpint) over `k` (a number) and returns the result as a new mpint.

11.4.4 Bitwise Operations

`mp.andint (a, b)`

Conducts a bitwise-and, `` `a and b` ``, with `a`, `b` mpints, and returns the result as a new mpint.

`mp.orint (a, b)`

Conducts a bitwise-or, `` `a or b` ``, with `a`, `b` mpints, and returns the result as a new mpint.

`mp.xorint (a, b)`

Conducts a bitwise-xor, `` `a xor b` ``, with `a`, `b` mpints, and returns the result as a new mpint.

`mp.com (a)`

Computes the one's complement of mpint `a` and returns the result as a new mpint.

`mp.scan0 (a, n)`
`mp.scan1 (a, n)`

Scan mpint `a`, starting from bit `n` (a number, not an mpint), towards more significant bits, until the first 0 or 1 bit (respectively) is found. The functions return the index of the found bit. Bit positions start from 0.

`mp.popcount (a)`

Computes the population count of mpint `a`, i.e. the number of 1 bits in the binary representation, and returns the result a number.

`mp.mostsigbit (a)`

Returns the position of the most significant bit (msb) as a number, counting from bit position number 0. If all bits are cleared, i.e. zero, returns -1.

`mp.leastsigbit (a)`

Returns the position of the least significant bit (lsb) in a number, counting from bit position number 0. If all bits are cleared, i.e. zero, returns -1.

`mp.hamdist (a, b)`

Computes the hamming distance between mpint `a` and mpint `b`, i.e. the number of positions `a` and `b` have different bit values. The count will be returned as a number.

`mp.setbit (r, n)`

Sets bit `n` in mpint `r`. The function returns nothing. `n` is an integer position counting from 0.

`mp.getbit (r, n)`

Returns bit `n` in mpint `r`. The function returns a number. `n` is an integer position counting from 0.

`mp.clrbit (r, n)`

Clears bit `n` in mpint `r`. The function returns nothing. `n` is an integer position counting from 0.

`mp.combit (r, n)`

Conducts a bitwise complement (`` `not` `` operation) on bit number `n` in mpint `r`. `n` is an integer position counting from 0. The function returns nothing.

11.4.5 Miscellaneous

`mp.tonumber (a)`

Returns the numeric value in mpint `a` as a number.

`mp.tostring (a)`

Returns the numeric value in mpint `a` as a string.

`mp.swap (a, b)`

Swaps the values in mpints `a` and `b`. The function returns nothing.

`mp.cmp (a, b)`

Compares `a` and `b` and returns a positive number if `a > b`, 0 if `a = b`, and a negative number if `a < b`.

`mp.cmpabs (a, b)`

Compares the absolute values of `a` and `b` and returns a positive number if `a > b`, 0 if `a = b`, and a negative number if `a < b`.

`mp.iseven (a)`

Checks whether mpint `a` represents an even integer and returns **true** or **false**.

`mp.isodd (a)`

Checks whether mpint `a` represents an odd integer and returns **true** or **false**.

`mp.setstring (str)`

Receives a string `str` and converts it to an mpint.

See also: **mp.getstring**, **mp.uint**, **mp.sint**.

`mp.getstring (a)`

Returns the string in mpint `a` that was previously stored to it by calling the **mp.setstring** function.

See also: **mp.setstring**, **mp.tostring**.

`mp.sizeinbase (a)`

Returns the size of `a` measured in number of digits in the given `base`, an integer. `base` can vary from 2 to 62 and should be even.

`mp.attrib (a)`

Returns various information on mpint a, in a dictionary.

11.5 mpf - GNU Multiple Precision Floating-Point Reliable Library

As a *plus* package, the **mpf** package is not part of the standard distribution and must be activated with the **import** statement, i.e. `import mpf`.

The mpf library is a binding to the MPFR library providing multiple functions to conduct floating-point arithmetic of arbitrary precision.

The package also provides various metamethods for easy entry of calculations.

MPFR floats are represented by mpf userdata objects which can be passed to the functions and operators described below.

OS/2, Solaris, Linux and UNIX users may have to install the original GMP 6.1 and MPFR 4.1 libraries separately in order for this binding to work. The package is not available for Mac OS X. In order for this binding to work on Intel CPUs, you may need at least a Sandybridge processor.

Examples:

```
> import mpf

> a, b := mpf.new(2), mpf.new(3)  # define 2.0 & 3.0 as two MPFR floats

> a * b:                 # conduct multiplication
mpf(6.0000000000000000000000000000000000000000)

> sin(mpf.Pi):           # compute sine of Pi
mpf(1.8830410776607851167459095484560349402730e-39)

> mpf.precision(32);     # reduce precision

> sin(mpf.Pi):
mpf(1.8830410775e-39)

> mpf.log2(mpf.new(8)):  # compute base-2 logarithm of 8
mpf(3.0000000000)
```

Following are all available operators, mathematical and general functions.

Operator	Functionality	Operator	Functionality
+	addition	exp	exponential function to the base E = 2.71828..
-	subtraction	sin	sine
*	multiplication	cos	cosine
/	division	tan	tangent
^	exponentiation	sinh	hyperbolic sine
(-)	unary minus	cosh	hyperbolic cosine
abs	absolute value	tanh	hyperbolic tangent
sign	sign	arcsin	inverse sine
sqrt	square root	arccos	inverse cosine
ln	natural logarithm	arctan	inverse tangent

Operator	Functionality
isnonzero	test for a non-zero
iszero	test for zero
=	equality check
<	less-than relation
<=	less-than-or-equal relation
>	greater-than relation
>=	greater-than or equal relation

Function	Arguments	Functionality
mpf.agm	2 MPFR values	arithmetic-geometric mean
mpf.ai	1 MPFR value	Airy function
mpf.arctan2	2 MPFR values	inverse tangent
mpf.beta	2 MPFR values	Beta function
mpf.recsqrt	1 MPFR value	inverse square root
mpf.cbrt	1 MPFR value	cubic root
mpf.ceil	1 MPFR value	rounds up to the next higher or equal representable integer
mpf.copysign	2 MPFR values	like **math.copysign**
mpf.cot	1 MPFR value	cotangent
mpf.csc	1 MPFR value	cosecant
mpf.digamma	1 MPFR value	Digamma function
mpf.dim	2 MPFR values a, b	returns a-b if a > b, 0 if a <= b, or undefined of a or b is undefined
mpf.eint	1 MPFR value	exponential integral
mpf.erf	1 MPFR value	error function
mpf.erfc	1 MPFR value	complementary error function
mpf.exp10	1 MPFR value	exponential to base 10
mpf.exp2	1 MPFR value	exponential to base 2
mpf.floor	1 MPFR value	rounds to the next lower or equal representable integer
mpf.fma	3 MPFR values	fused multiply-addition
mpf.fmod	2 MPFR values	see: **fmod**
mpf.fms	3 MPFR values	fused multiply-subtraction
mpf.gamma	1 MPFR value	Gamma function
mpf.hypot	2 MPFR values	hypotenuse
mpf.isfinite	1 MPFR value	check for a finite value, unlike undefined or infinity
mpf.isinfinite	1 MPFR value	check for infinity
mpf.isundefined	1 MPFR value	check for **undefined**
mpf.j0	1 MPFR value	first kind Bessel function of order 0
mpf.j1	1 MPFR value	first kind Bessel function of order 1
mpf.jn	2 MPFR values plus one integer for the order	first kind Bessel function of order n

Function	Arguments	Functionality
mpf.y0	1 MPFR value	second kind Bessel function of order 0
mpf.y1	1 MPFR value	second kind Bessel function of order 1
mpf.yn	2 MPFR values plus one integer for the order	second kind Bessel function of order n
mpf.li2	1 MPFR value	real part of the dilogarithm of its argument
mpf.lgamma	1 MPFR value	logarithm of the Gamma function
mpf.log10	1 MPFR value	logarithm to the base 10
mpf.log2	1 MPFR value	logarithm to the base 2
mpf.modf	2 MPFR values	see: **modf**.
mpf. nexttoward	1 MPFR value	works like `math.nextafter`, but for MPFR values; does not change its argument
mpf.random	none	returns a uniformly distributed random float on the interval [0, 1]
mpf.round	1 MPFR value	rounds to nearest representable integer, rounding halfway cases away from zero
mpf.sec	1 MPFR value	secant
mpf.signbit	1 MPFR value	checks the sign bit and returns **true** (value is negative) or **false**
mpf.trunc	1 MPFR value	rounds to the next representable integer toward zero
mpf.zeta	1 MPFR value	Riemann Zeta function
mpf.Zero	1 signed integer	returns an MPFR +0 or -0, depending on sign of its argument
mpf.Inf	1 signed integer	returns an MPFR +**infinity** or -**infinity**, depending on sign of its argument
mpf.Nan	none	returns MPFR **undefined**
mpf.max	2 MPFR values	returns the maximum of two values
mpf.min	2 MPFR values	returns the minimum of two values
mpf.Ln2	n/a	MPFR constant ln(2)
mpf.Pi	n/a	MPFR constant Pi
mpf.Euler	n/a	MPFR constant $\gamma = 0.57721566490...$
mpf.Catalan	n/a	MPFR constant $\lambda = 0.91596559417...$

See also: The **long** package implementing 80-bit floating-point arithmetic, described in Chapter 11.15.

General functions:

`mpf.clone (x)`

Clones an MPFR value and returns it. The rounding mode of the MPFR value returned will be the current one, not necessarily the one with which the value to be duplicated has been created.

`mpf.new (x)`

Creates an MPFR floating-point object from a number x, or a string x representing a number. For best accuracy, you should pass strings instead of numbers, as numbers are rounded before converted to an MPFR value.

`mpf.precision ([x])`

Gets or sets the overall precision, in bits. If an integer x in the range 2 .. 2,147,483,647 is being passed, the function sets the precision for all values subsequently allocated.

If no argument is given, the current setting will be returned.

The default precision at invocation of the package is 128.

`mpf.rounding ([rmode])`

Gets or sets the current rounding mode. If a string `rmode` is passed, the function sets the rounding mode for all values subsequently allocated. Valid settings for `rmode` are the strings:

- `'rndn'`, round to nearest, with ties to even;
- `'rndz'`, round toward zero;
- `'rndu'`, round toward +infinity;
- `'rndd'`, round toward -infinity.

If no argument is given, the current rounding mode will be returned.

The default rounding mode at invocation of the package is `'rndn'`, i.e. rounding to nearest.

`mpf.swap (x, y)`

Swaps the values in MPFR values x and y. The function returns nothing.

`mpf.tonumber (x)`

Converts an MPFR value x into an Agena number.

`mpf.tostring (x)`

Converts an MPFR value x into a string.

11.6 divs - Library to Process Fractions

As a *plus* package, this library is not part of the standard distribution and must be activated with the **import** statement, e.g. `import divs`.

The library provides basic arithmetic to calculate with fractions. To create a fraction, use **divs.divs** which accepts mixed, improper and proper fractions. The package implements metamethods so that the common addition, subtraction, division, and unary minus operators can be used.

The **+** operator adds two fractions, or a number and a fraction in any order.

The **-** operator subtracts two fractions, or a number and a fraction in any order.

The ***** operator multiplies two fractions, or a number and a fraction in any order.

The **/** operator divides two fractions, or a number and a fraction in any order.

The **^** operator exponentiates two fractions, or a number and a fraction in any order.

The ****** operator raises a fraction to an integer power, in this order.

The **abs** operator returns the absolute value of a fraction and returns a fraction.

The **sign** operator returns the sign of a fraction and returns a number.

The **sqrt** operator returns the square root of a fraction and returns a fraction. If the resulting fraction could not be evaluated with absolute precision, it returns a number.

The **ln** operator returns the natural logarithm of a fraction and returns a fraction. If the resulting fraction could not be evaluated with absolute precision, it returns a number.

The **exp** operator returns the value of **E** to the power of the given fraction and returns a fraction. If the resulting fraction could not be evaluated with absolute precision, it returns a number.

The **sin** operator returns the sine of a fraction and returns a fraction in radians. If the resulting fraction could not be evaluated with absolute precision, it returns a number (in radians).

The **cos** operator returns the cosine of a fraction and returns a fraction in radians. If the resulting fraction could not be evaluated with absolute precision, it returns a number (in radians).

The **tan** operator returns the tangent of a fraction and returns a fraction in radians. If the resulting fraction could not be evaluated with absolute precision, it returns a number (in radians). It returns **undefined** if poles have been encountered.

The **arctan** operator returns the arcus tangent of a fraction and returns a fraction in radians. If the resulting fraction could not be evaluated with absolute precision, it returns a number (in radians). It returns **undefined** if poles have been encountered.

The **int** operator returns the integer quotient of the numerator of a fraction divided by its denominator.

The numerators and denominators should all be integers.

The return always is an improper fraction. There are also two functions to convert fractions to decimals and vice versa.

Examples:

```
> import divs;

> divs.divs(1, 2, 3) + divs.divs(1, 3):
2

> divs.divs(1, 2) * divs.divs(1, 3):
divs(5, 6)

> divs.divs(1, 2) * divs.divs(1, 3):
divs(1, 6)

> 2 * divs.divs(1, 3):
divs(2, 3)

> divs.todec(divs.divs(1, 2)):
0.5

> divs.todiv(ans):
div(1, 2)          0
```

Relations: Two fractions can be compared with the <, <=, =, ==, ~=, >=, and > operators.

The following operators are also supported: **arcsin**, **arccos**, **arcsec**, **sinh**, **cosh**, **tanh**, **recip**, and ~<>.

Functions:

divs.denom (a)

This function returns the denominator of the fraction `a` of the user-defined type `'divs'` and returns it as a number.

The function is written in Agena and is included in the lib/divs.agn file.

See also: **divs.numer**.

divs.divs ([x,] y, z)

divs.divs ([x:]y:z)

This function defines a fraction and returns it as a value of the user-defined type `'div'` if `z` is not 1, with proper metamethods added. It returns a number if `z` equals 1, and **undefined** if `z` is 0.

In the first form: if all three arguments are given, representing a mixed fraction $x \frac{y}{z}$, the function converts it into an improper fraction and returns it. If only `y` and `z` are given, the function returns a reduced improper or proper fraction $\frac{x}{y}$.

The second form allows to pass `x`, `y`, and `z` as a nested pair `x:y:z`, representing a mixed fraction, or the pair `y:z` representing an improper or proper fraction.

In both forms, `x`, `y`, and `z` should be integers.

The function is written in Agena and is included in the lib/divs.agn file.

divs.equals (a, b [, option])

This function checks two fractions `a`, `b` for equality. Alternatively, either `a` or `b` may be simple Agena numbers. The result is either **true** or **false**. If any non-**null** `option` is given, the function checks for approximate equality (see **approx** function). Note that the equality operators =, ==, and ~= cannot check values of different types.

The function is written in Agena and is included in the lib/divs.agn file.

divs.numer (a)

This function returns the numerator of the fraction `a` of the user-defined type `'divs'` and returns it as a number.

The function is written in Agena and is included in the lib/divs.agn file.

See also: **divs.denom**.

`divs.todec (a)`

This function converts a fraction `a` of the user-defined type `'divs'` to a float and returns it.

The function is written in Agena and is included in the lib/divs.agn file.

See also: **divs.todiv**.

`divs.todiv (x)`

This function converts a number `x` to an improper fraction of the user-defined type `'divs'` and returns it. The second return is the accuracy (see **math.fraction** for further information).

The function is written in Agena and is included in the lib/divs.agn file.

See also: **divs.todec**, **math.fraction**.

11.7 dual - Dual Numbers

As a *plus* package, the **dual** package is not part of the standard distribution and must be activated with the **import** statement, i.e. `import dual`.

This library provides basic support for dual numbers which are related to complex numbers, but instead of an imaginary unit i with $i^2 = -1$, we have a nilpotent ε unit with $\varepsilon^2 = 0$.

Dual numbers are used with automatic differentiation, and other applications.

The package provides basic arithmetic operators via and also some transcendent functions, in cases, through metamethods.

To define a dual number, e.g. $1+2\varepsilon$, type:

```
> import dual

> a := dual.dual(1, 2)

> a:
1+2e
```

Add a to $3+4\varepsilon$:

```
> b := dual.dual(3, 4)

> a+b:
4+6e
```

Square root:

```
> sqrt(ans):
2+1.5e
```

The following lists all available operators and functions and the results, with two dual numbers $p = a + b\varepsilon$ and $q = c + d\varepsilon$:

Operation	Call	Real Part	Dual Part
Unary minus	-p	-a	-b
Addition	p + q	a + c	b + d
Subtraction	p - q	a - c	b - d
Multiplication	p * q	a * c	a*d + b*c
Division	p / q	a / c	(b*c - a*d) / (c*c)
Reciprocal	1 / p	1 / a	-b/(a**2)
Exponentiation	p ^ q	a ^ c	a ^ c*(d*ln a + b*c/a)
Square	square p	a**2	2*a*b
Cube	cube p	a**3	3*b*a**2
Absolute value	abs p	abs a	n/a
Sign	sign p	1, if a > 0 or a = 0 and b > 0 -1, if a < 0 or a = 0 and b < 0 0, otherwise	n/a
Exponential function	exp p	exp a	b*exp a
Natural logarithm	ln p	ln a	b/a
Base-2 logarithm	log2	log2(a)	b/a/log(2)
Base-10 logarithm	log10	log10(a)	b/a/log(10)
exp(x)-1	exp-minusone	expminusone(a)	b*exp(a)
ln(x+1)	lnplusone	lnplusone(a)	b/(1.0 + a)
Square root	sqrt p	sqrt a	0.5*a[2]/sqrt a
Hypotenuse	hypot p	hypot(a, b)	./.
Sine	sin p	sin a	b*cos a
Cosine	cos p	cos a	-b*sin a
Tangent	tan p	tan a	b/(cos(a)**2)
Arcus sine	arcsin p	arcsin a	b/hypot3(a)
Arcus cosine	arccos p	arccos a	-b/hypot3(a)
Arcus tangent	arctan p	arctan a	b/(1 + a*a)
Hyperbolic sine	sinh p	sinh a	b*cosh a
Hyperbolic cosine	cosh p	cosh a	b*sinh a
Hyperbolic tangent	tanh p	tanh a	b*sech(a) ^ 2
Inverse hyperbolic sine	arcsinh	arcsinh(a)	b*(1/sqrt(a ^ 2 + 1))
Inverse hyperbolic cosine	arccosh	arccosh(a)	b*(1/sqrt(a ^ 2 - 1))
Inverse hyperbolic tangent	arctanh	arctanh(a)	b*(1/(1 - a ^ 2))
Error function	erf	erf(a)	b*2/sqrt(Pi) * exp(-(a) ^ 2)
Complementary error function	erfc	erfc(a)	b*-2/sqrt(Pi)*exp(-(a) ^ 2)
Scaled complementary error function	erfcx	erfcx(a)	2*a * expx2(a) * erfc(a) - 2/sqrt(Pi)
Equality	p = q	a = c and b = d	n/a

Operation	Call	Real Part	Dual Part
Relation <	p < q	a < c	n/a
Relation <=	p <= q	a <= c	n/a
Relation >	p > q	a > c	n/a
Relation >=	p >= q	a >= c	n/a

11.8 clock - Clock Package

This package contains mathematical routines to perform basic operations on time values, i.e. hours, minutes, and seconds.

As a *plus* package, it is not part of the standard distribution and must be activated with the **import** statement, e.g. `import clock`.

A time value is always defined by the **clock.tm** constructor. You may apply the ordinary `+`, `-`, `*` and / operators in order to add, subtract, multiply or divide values. The relations <, <=, =, >=, and > are also supported.

Also, the following operators can be used for sexagesimal arithmetic - but please beware of round-off errors, for they convert a sexagesimal argument to decimal, apply the operator, and convert the result back to sexagesimal.

The `^` operator exponentiates sexagesimals, or sexagesimals and numbers, and returns a sexagesimal.

The **abs** operator determines the absolute value of a sexagesimal and returns a sexagesimal.

The **sign** operator returns the sign of a sexagesimal and returns a number.

The **sqrt** operator returns the square root of a sexagesimal and returns a sexagesimal. If the sexagesimal is negative, it returns **undefined**.

The **ln** operator returns the natural logarithm of a sexagesimal and returns a sexagesimal. If the sexagesimal is non-negative, it returns **undefined**.

The **exp** operator returns the value of **E** to the power of the given sexagesimal and returns a sexagesimal.

The **sin** operator returns the sine of a sexagesimal and returns a sexagesimal, in radians.

The **cos** operator returns the cosine of a sexagesimal and returns a sexagesimal, in radians.

The **tan** operator returns the tangent of a sexagesimal and returns a sexagesimal, in radians. It returns **undefined** if poles have been encountered.

The **arctan** operator returns the arcus tangent of a sexagesimal and returns a sexagesimal, in radians. With poles, it returns **undefined**.

By default, all time values are properly adjusted to a normalised representation if the value of the environment variable **_clockAdjust** is not changed. If it

_clockAdjust is set to a value different from **true**, then this normalisation is switched off.

All functions are implemented in Agena and included in the lib/clock.agn file.

A typical example might look like this:

```
> import clock alias

add, adjust, div, mul, sub, pow, tm, todec, totm
```

Subtract 10 hours and fifteen minutes from 20 hours and 15 minutes:

```
> tm(20, 15, 0) - tm(10, 15, 0):
tm(10, 0, 0)
```

61 seconds are automatically converted to 1 minute and 1 second:

```
> tm(0, 61):
tm(0, 1, 1)
```

Turn off normalisation:

```
> _clockAdjust := null

> tm(0, 61):
tm(0, 0, 61)
```

Turn on normalisation again:

```
> _clockAdjust := true
```

The functions provided by the package are:

clock.add (t1, t2 [, ···])
The function adds two or more values of type **tm**. The return is a value of type **tm**.

clock.adjust (t)
The function adjusts the representation of tm values in a time object `t` by applying the rules described in the description of **clock.tm**.

clock.sub (t1, t2 [, ···])
The function subtracts two or more values of type **tm**. The return is a value of type **tm**.

`clock.sgstr (x [, d])`

Converts a float or `tm` value `x` into its sexagesimal string representation of the format hh:mm:ss. The colon to separate hours, minutes, and seconds can be changed by passing another optional delimiter `d` of type string.

See also: **clock.totm**.

`clock.tm (min)`

`clock.tm (min, sec)`

`clock.tm (hrs, min, sec)`

This function is used to define time values, where `hrs`, `min`, `sec` are numbers.

In the first form, minutes are defined. The return is a value of type tm of the form tm(0, min, 0).

In the second form, both minutes and seconds are defined. The return is a value of type tm of the form tm(0, min, sec).

In the third form, both hours, minutes, and seconds are defined and returned as a value of type tm of the form tm(hrs, min, sec). (hrs may be set to 0.)

By default, if min > 59 and / or if sec > 59, proper adjustments are made before the time value will be returned. If min > 59 the call to **time** returns tm(hrs + 1, min - 60, sec). If sec > 59 the call to time returns tm(hrs, min + 1, sec - 60). The default is set by the global variable `_clockAdjust` which is assigned **true** at initialisation of the package if it has not already been set **false** before the clock package has been loaded.

`hrs` might be any non-negative number.

If `_clockAdjust` is set false then no adjustments are made to the arguments. You can use **clock.adjust** to apply the adjustments described above.

`clock.todec (t)`

Converts a tm value `t` into its decimal representation of type number.

See also: **clock.totm**, **math.todecimal**.

`clock.totm (t)`

Converts a tm value `t` in decimals (of type number) into its tm representation. The return is of type tm.

See also: **clock.todec**.

11.9 astro - Astronomy Functions

As a *plus* package, the **astro** package is not part of the standard distribution and must be activated with the **import** statement, e.g. `import astro`.

`astro.cdate (x)`

Converts a Julian date, represented by the float `x`, into its Gregorian calendar date representation, returning seven values in the following order: the year, the month, the day, the decimal fraction of the day - in the range [0, 1) -, the hour, minute, and second.

See also: **astro.jdate**, **os.date**, **os.isdst**.

`astro.cweek (y, m, d)`

Returns the calendar week for the given year (`y`), month (`m`), and day (`d`), an integer in the range 1 .. 53, determined according to the ISO 8601 standard, and the corresponding year as the second result. The second result is not necessarily equal to `y`, e.g. the calendar week of January 1, 2016 is calendar week 53 of 2015, and the calendar week of December 31, 2013 is week 1 of 2014.

If the passed date does not exist, the function issues an error.

See also: **astro.lastcweek**, **os.date**.

`astro.cweekmonsun (y, cw)`

Computes the Gregorian dates of the Monday and Sunday for a given calendar week `cw` and year `y` and returns four numbers: the year, the month, the day, and the fraction of day - in his order. In case of a non-existing calendar week `cw`, the function issues an error.

See also: **astro.cweek**, **os.date**.

`astro.dectodms (x, orientation)`

Converts co-ordinates `x` in decimal degrees (a number) to the form degree, minute, second, and their orientation 'N', 'S', 'W', or 'E' (DMS format). You must also specify whether to compute latitude or longitude values, by passing the strings `'lat'` or `'lon'`, respectively for `orientation`.

The return are three numbers and the orientation, a string.

See also: **astro.dmstodec**.

`astro.dmstodec (degree, minute, second, hour, orientation)`

Converts co-ordinates in DMS format consisting of `degree`, `minute`, `second`, (all numbers) and their `orientation` 'N', 'S', 'W', or 'E' (a single-character string) to their corresponding decimal degree representation (DegDec format). The return is a number.

See also: **astro.dectodms**.

`astro.hdate (jd)`

Converts the Julian date `jd` to the corresponding year, month and day in the Jewish calendar, in this order. The fraction of day, the hour, minute and second are also returned.

See also: **astro.jdate**, **os.date** with the `'*j'` format.

`astro.isleapyear (x)`

Returns **true** if the given year `x` (a number) is a leap year, and **false** otherwise.

`astro.lastcweek (y [, flag])`

Computes the last calendar week of the year `y`. If `flag` is not given, the function returns either the number 52 or 53, and the given year `y`. If `flag` is given, then calendar week 1 and `y + 1` will be returned if December 31 of the given year `y` is either a Monday, Tuesday or Wednesday, otherwise it works as if `flag` has not been passed.

See also: **astro.lastcweek**, **os.date**.

`astro.jdate (year, month, day [, hour [, minute [, second]]])`

Converts a Gregorian date represented by `year`, `month`, `day` and optionally `hour`, `minute`, and `second` (all numbers) to the corresponding Julian date. The return is a number, or **fail** if the date or time is of a wrong format.

By definition, the base 0 of the Julian date is January 1, 4713 BC, noon GMT. However, since the function takes no account of the date(s) of adoption of the Gregorian calendar, astro.jdate(0) does not return this date.

The defaults for `hour`, `minute`, and `second` are 0.

See also: **astro.cdate**, **os.date**, **os.isdst**.

`astro.moon (year, month, day, hour, lon, lat)`

Provides an easier-to-use interface to **astro.moonriseset**. and **astro.moonphase**.

The first four arguments represent the `year`, `month`, `day`, and `hour`, all of type number. Longitudes and latitudes can be given in form of two tables `lon`, `lat` containing degrees (a number), minutes (a number), seconds (a number), and the orientation (the single character 'N', 'S', 'W', or 'E').

The return is a table with the indices 'riseset', containing the rise and set times of the Moon in `tm` representation, and the index 'phase' which holds the computed Lunar phase (a float and an integer).

See **astro.moonriseset** and **astro.moonphase** for further information.

The function uses the `tm` time notation of the **clock** package. You do not have to readlib **clock** before.

The function is written in Agena and included in the `astro.agn` file.

Example for Düsseldorf:

```
> astro.moon(2013, 1, 7, 0, [7, 6, 0, 'E'], [50, 43, 48, 'N']):
[phase ~ [0.2995659104481, 7], riseset ~ [tm(2, 27, 0), tm(11, 50, 0)]]
```

astro.moonphase (year, month, day [, hour])

Takes a `year`, a month, a `day`, and optionally an `hour` (all numbers) and returns the moon phase as a real number in the range [0, 1], where 0 is new moon and 1 is full Moon; and an integer in the range [0, 7], where 0 indicates new moon and 4 indicates full moon. If hour is not given, it is set to 0.

See also: **astro.moon**.

astro.moonriseset (year, month, day, lon, lat)

Returns the times of Lunar rise and set in GMT. Receives the `year`, `month day`, the longitude and latitude `lon` and `lat` (all of type number) and returns two numbers: the GMT rise time in a decimal, and the GMT set time also in a decimal.

Use **clock.totm** to convert the rise and set times to sexagesimal format, or try **astro.moon**.

Example for Düsseldorf:

```
> astro.moonriseset(2013, 1, 8,
>   astro.dmstodec(6, 46, 58, 'E'), astro.dmstodec(51, 13, 32, 'N')):
3.7666666666667 12.566666666667
```

astro.sun (year, month, day, lon, lat)

Provides an easier-to-use interface to **astro.sunriseset**.

`year`, `month`, and `day` must be integers. Longitudes and latitudes can be given in form of two tables `lon`, `lat`, containing degrees (a number), minutes (a number), seconds (a number), and the orientation (the single-character string 'N', 'S', 'W', or 'E').

The return is a table with the indices 'riseset', 'civil', 'astro', and 'nautical' containing the rise and set times in `tm` representation. The index 'south' holds the time where the Sun is at south.

See **astro.sunriseset** for further information.

The function uses the `tm` time notation of the **clock** package. The function uses the `tm` time notation of the **clock** package. You do not have to readlib **clock** before.

The function is written in Agena and included in the `astro.agn` file.

Example for Düsseldorf:

```
> astro.sun(2013, 1, 7, [6, 46, 58, 'E'], [51, 13, 32, 'N']):
[astro ~ [tm(5, 34, 5.1483689555826), tm(17, 44, 22.952745470386)],
civil ~ [tm(6, 56, 25.738372228174), tm(16, 22, 2.3627421977944)],
nautical ~ [tm(6, 14, 13.023074498407), tm(17, 4, 15.078039927568)],
riseset ~ [tm(7, 35, 19.775508661645), tm(15, 43, 8.325605764323)],
south ~ tm(11, 39, 14.050557212984)]
```

astro.sunriseset (year, month, day, lon, lat)

Returns the sunrise/sunset times in UTC for years starting with 1800 A.D. to 2099 A.D. It is a workhorse function, maybe you would like to use **astro.sun** for a more convenient interface.

`year`, `month` and `day`, all integers, are the values of the day to evaluate. `lon` is the longitude (west/east), and `lat` the latitude (west/east), both in decimal degrees of type float of the location that is of interest. Use **astro.dmstodec** to convert co-ordinates containing degrees (integer), minutes (integer), and seconds (integer or float), and the orientation to decimal degrees.

Example for Düsseldorf:

```
> astro.sunriseset(2013, 1, 7,
>   astro.dmstodec(6, 46, 58, 'E'), astro.dmstodec(51, 13, 32, 'N')):

7.5888265301838 15.718979334935 0    6.9404828811745 16.367322983944 0
6.2369508540273 17.070855011091 0    5.5680967691543 17.739709095964 0
11.653902932559
```

The first and second returns are the sunrise/sunset times which are considered to occur when the Sun's upper limb is 35 arc minutes below the horizon (this accounts for the refraction of the Earth's atmosphere).

The third return is 0, if the rises and sun sets in a day; +1 if the Sun is above the specified `horizon` 24 hours, -1 if the Sun is below the specified `horizon` 24 hours.

The fourth and fifth returns are start and end times of civil twilight. Civil twilight starts/ends when the Sun's centre is 6 degrees below the horizon.

The sixth return is 0, if the rises and sun sets in a day; +1 if the Sun is above the specified `civil twilight horizon` 24 hours, -1 if the Sun is below the specified `horizon` 24 hours.

The seventh and eighth returns are the start and end times of nautical twilight. Nautical twilight starts/ends when the Sun's centre is 12 degrees below the horizon.

The ninth return is 0, if the rises and sun sets in a day; +1 if the Sun is above the specified `nautical twilight horizon` 24 hours, -1 if the Sun is below the specified `horizon` 24 hours.

The tenth and eleventh returns are the start and end times of astronomical twilight. Astronomical twilight starts/ends when the Sun's centre is 18 degrees below the horizon.

The twelfth return is 0, if the rises and sun sets in a day; +1 if the Sun is above the specified `nautical twilight horizon` 24 hours, -1 if the Sun is below the specified `astronomical twilight horizon` 24 hours.

The thirteenth return is the time when the Sun is at south (in decimal UTC).

All times returned are given in decimal hours of type number. Use **clock.totm** to convert them into `tm` notation.

See also: **astro.sun**, **astro.moon**.

`astro.taiutc ([jd])`

Returns the TAI-UTC lookup table value of leap seconds for a given Julian date `jd`; if no argument is given, then the value for the current system date is computed. TAI stands for International Atomic Time. The function returns 0 for Gregorian dates before 1961.

In the future, you may have to add further values to the source code of this function which also includes the URL of the respective file to be checked. The function is written in Agena and included in the `astro.agn` file.

See also: **os.date** ('*j' format).

11.10 cordic - Numerical CORDIC Library

As a *plus* package, this library is not part of the standard distribution and must be activated with the **import** statement, e.g. `import cordic`.

The CORDIC algorithm (CORDIC stands for COordinate Rotation DIgital Computer) also known as the `Volder's algorithm`, is used to calculate hyperbolic, trigonometric, logarithmic, and root functions, on hardware not featuring multipliers, requiring only addition, subtraction, bitshift and table lookup.

The algorithm, similar to one published by Henry Briggs around 1624, has been developed in 1959 by Kack E. Volder to improve an aviation system. According to Wikipedia, it has not only been used in pocket calculators, but also in x87 FPUs, in CPUs prior to Intel 80486 - and in Motorola's 68881, in signal and image processing, communication systems, robotics, and also 3D graphics - and other applications.

This binding to John Burkardt's CORDIC implementation uses additon, subtraction, table lookups, multiplication, divisions, and the absolute function.

The package accepts and returns Agena numbers only.

Available functions are:

cordic.carccos (x)

Returns the inverse cosine operator in radians.

cordic.carcsin (x)

Returns the inverse sine operator in radians.

cordic.carctan2 (y, x)

Returns the arc tangent of y/x in radians, but uses the signs of both parameters to find the quadrant of the result.

cordic.carctanh (x)

Returns the inverse hyperbolic tangent of x in radians.

cordic.ccbrt (x)

Returns the cubic root of the number x.

cordic.ccos (x)

Returns the cosine of x in radians.

cordic.ccosh (x)

Returns the hyperbolic cosine of x in radians.

cordic.cexp (x)

Returns e^x, the exponential function to the base $e = 2.718281828459$ ...

cordic.chypot (x, y)

Returns $\sqrt{x^2 + y^2}$, the hypotenuse.

cordic.cln (x)

Returns the natural logarithm of x.

cordic.cmul (x, y)

Returns $x*y$, i.e. the product of x and y.

cordic.csin (x)

Returns the sine of x in radians.

cordic.csinh (x)

Returns the hyperbolic sine of x in radians.

cordic.csqrt (x)

Returns the square root of x.

cordic.ctan (x)

Returns the tangent of x in radians.

cordic.ctanh (x)

Returns the hyperbolic tangent of x in radians.

11.11 zx - Sinclair ZX Spectrum Functions

As a *plus* package, the **zx** package is not part of the standard distribution and must be activated with the **import** statement, i.e. `import zx`.

11.11.1 Introduction

This package implements various Sinclair ZX Spectrum mathematical functions.

Most of the functions use the same algorithms and Chebyshev polynomials of degree 6, 8, or 12 as implemented in the Sinclair ZX Spectrum ROM, with similar accuracy.

All functions are based on those published on the book `The Complete Spectrum ROM Disassembly`, written by Dr. Ian Logan & Dr. Frank O'Hara, pp. 217.

In general, the procedures are mostly slower and also less precise than their Agena pendants. By default, the fully expanded and simplified polynomials are hard-wired into the library's C code. By passing the optional last argument **true**, however, the polynomials are processed iteratively in real-time, using the **zx.genseries** function which imitates the Z80 assembler subprocedure `series generator`.

You may query the respective Chebyshev coefficient vectors by calling **zx.getcoeffs**, and globally change them with **zx.setcoeffs**. Range reduction is performed by **zx.reduce**.

The names of all ZX Spectrum `clones` are written in capital letters, to not collide with Agena's built-in operators.

The C source file src/zx.c contains exact information on the precision of the functions.

11.11.2 Original ZX Spectrum Functions

`zx.ABS (x)`

Returns the absolute magnitude of the number `x`. The function does not use Chebyshev polynomials.

See also: **abs**.

`zx.ACS (x)`

Computes the ZX Spectrum inverse cosine of its numeric argument x and returns a number. If $x \notin [-1, 1]$, **undefined** will be returned.

See also: **arccos**.

`zx.ADD (x, y)`

Returns $x + y$.

See also: **zx.DIV**, **zx.MUL**, **zx.SUB**.

`zx.AND (x, y)`

Returns x if y is non-zero and the value zero otherwise. Strings are not supported. The function does not use Chebyshev polynomials.

See also: **zx.NOT**, **zx.OR**.

`zx.ASN (x)`

Computes the ZX Spectrum inverse sine of its numeric argument x and returns a number. If $x \notin [-1, 1]$, **undefined** will be returned.

See also: **arcsin**.

`zx.ATN (x)`

Computes the ZX Spectrum inverse tangent of its numeric argument x and returns a number.

See also: **arctan**.

`zx.COS (x)`

Computes the ZX Spectrum cosine of its numeric argument x and returns a number.

See also: **cos**.

`zx.DIV (x, y)`

Returns x / y. With $y = 0$, the result is **undefined**.

See also: **zx.ADD**, **zx.MUL**, **zx.SUB**.

`zx.EXP (x)`

Computes the ZX Spectrum exponential function of the number x to the base **E** = exp(1). It looses precision, however, if its argument is greater than the constant **E**.

See also: **exp**. **zx.LN**.

`zx.INT (x)`

Rounds its numeric argument x downwards to the nearest integer. The function does not use Chebyshev polynomials.

See also: **entier**.

`zx.LN (x)`

Computes the ZX Spectrum natural logarithm of the number x. If $x \leq 0$, **undefined** will be returned.

See also: **ln**, **zx.EXP**.

`zx.MUL (x, y)`

Computes $x*y$.

See also: **zx.ADD**, **zx.DIV**, **zx.SUB**.

`zx.NOT (x)`

Returns 1 if its numeric argument x is 0, and 0 otherwise. The function does not use Chebyshev polynomials.

See also: **not**, **zx.AND**, **zx.OR**.

`zx.OR (x, y)`

Returns the number x if the number y is 0, and 1 otherwise. Strings are not supported. The function does not use Chebyshev polynomials.

See also: **or**, **zx.AND**, **zx.NOT**.

`zx.PI`

The constant π in the ZX Spectrum precision (supposedly C float).

`zx.POW (x, y)`

Returns the ZX Spectrum exponentiation $x \uparrow y$, with x and y numbers, and returns a number.

Internally, the ZX Spectrum and this function treats x $\uparrow$ y like exp(ln(x)*y). If $x < 0$ then **undefined** will be returned.

As with **zx.EXP**, the function is quite imprecise if $x > E$ (the constant).

See also: ****** and **^** operators, **zx.SQR**.

`zx.SGN (x)`

Returns -1 if the number x is negative, 0 if x is zero, and 1 if x is positive. If x is **undefined**, **undefined** will be returned. The function does not use Chebyshev polynomials.

See also: **sign**, **signum**.

`zx.SIN (x)`

Computes the ZX Spectrum sine of its numeric argument x and returns a number.

See also: **sin**, **zx.COS**, **zx.TAN**.

`zx.SQR (x)`

Returns the ZX Spectrum square root of its numeric argument x and returns a number. If $x < 0$, **undefined** will be returned.

See also: **sqrt**, **zx.POW**.

`zx.SUB (x, y)`

Computes $x - y$.

See also: **zx.ADD**, **zx.DIV**, **zx.MUL**.

`zx.TAN (x)`

Computes the ZX Spectrum tangent of its numeric argument x and returns a number.

See also: **tan**, **zx.COS**, **zx.SIN**.

11.11.3 Auxiliary Functions

`zx.genseries (x, s)`

Receives a number x in the range [-1, 1] and a sequence of coefficients and returns the value of the corresponding Chebyshev polynomial. If x is out of range, *no* error will be returned. This is an exact clone of the ZX Spectrum ROM `series generator` Z80 assembler subroutine.

`zx.getcoeffs ()`

The function returns the current Chebyshev coefficient vectors for various package functions. The return is a dictionary of four numeric sequences:

Key	Used by	Indirectly used by	Default size
`'SIN'`	**zx.SIN**	**zx.COS** and **zx.TAN**	6
`'ATN'`	**zx.ATN**	**zx.ACS** and **zx.ASN**	12
`'LN'`	**zx.LN**	**zx.SQR** and **zx.POW**	12
`'EXP'`	**zx.EXP**	**zx.SQR** and **zx.POW**	8

See also: **zx.setcoeffs**.

`zx.setcoeffs (n, s)`

Globally sets Chebyshev coefficients to the package's environment. You can change existing coefficients, reduce or enlarge their respective number down to one or up to 256 values. Internally, the coefficients are treated as C doubles, the shipped defaults have the precision of C floats.

The first argument n must be the string `'SIN'`, `'ATN'`, `'LN'`, or `'EXP'`. The second argument s must be a sequence of one to 256 numbers.

For the meanings of the first argument, see **zx.getcoeffs**.

Please note that the respective zx functions must be called with the last argument **true** in order to revert to the (changed) coefficient vectors as they use hard-wired expanded polynomials by default.

See also: **zx.getcoeffs**.

`zx.reduce (x)`

Reduces a number x to another number v in the range [-1, 1] where sin(v) = sin(π *v/2) for multiples or fractions of π, and returns v. Please note that even if $x \in$ [-1, 1], v will be calculated - see **inrange** for range checks.

The function imitates the ZX Spectrum ROM 'reduce argument' Z80 assembler subroutine which is used to prepare calls to ZX Spectrum's sine and cosine subroutines. Example:

```
> zxsin := proc(x) is
>    local w, z;
>    w := zx.reduce(x);
>    z := 2 * w**2 - 1;
>    return w * zx.genseries(z, zx.getcoeffs().SIN)
> end;

> zxcos := << x -> zxsin(x + Pi/2) >>;
```

See also: **math.wrap**, end of Chapter 11.1.2 for a comparison chart.

11.12 calc - Calculus Package

This package contains mathematical routines to perform basic calculus *numerically*. Since the functions do not work symbolically, please beware of round-off errors. As a *plus* package, it is not part of the standard distribution and must be activated with the **import** statement, e.g. `import calc`.

A typical example might look like this:

```
> import calc;
```

Define a function f :=x → sin(x):

```
> f := << x -> sin(x) >>
```

Determine all its zeros over [-5, 5]:

```
> calc.zeros(f, -5, 5):
seq(-3.1415926535898, 0, 3.1415926535898)
```

Differentiate it at point 0 and also return an error estimate:

```
> calc.diff(f, 0):
0.99999999999963          1.8503717573394e-010
```

Compare it:

```
> cos(0):
1
```

Integrate it over $[0, \pi]$:

```
> calc.gtrap(f, 0, Pi):
1.9999999938721
```

Summary of functions:

Basic Calculus:

> **calc.aitken, calc.fminbr, calc.fmings, calc.iscont, calc.limit, calc.regulafalsi, calc.sections, calc.zeroin, calc.zeros**.

Differentiation:

> **calc.diff, calc.eulerdiff, calc.isdiff, calc.maximum, calc.minimum, calc.xpdiff**.

Integration:

> **calc.gauleg, calc.gtrap, calc.intde, calc.intdei, calc.intdeo, calc.integ, calc.simaptive**.

Integrals:

> **calc.Ci, calc.Chi, calc.dawson, calc.Ei, calc.En, calc.fresnelc, calc.fresnels, calc.ibeta, calc.igamma, calc.igammc, calc.invibeta, calc.scaleddawson, calc.Shi, calc.Si, calc.Ssi, calc.w**.

Sums & Products:

> **calc.prod, calc.fsum**.

Interpolation:

> **calc.cheby, calc.chebyt, calc.chebycoeffs, calc.clampedspline, calc.clampedsplinecoeffs, calc.interp, calc.linterp, calc.nakspline, calc.naksplinecoeffs, calc.neville, calc.newtoncoeffs, calc.polyfit, calc.polygen, calc.savgol, calc.savgolcoeffs, calc.smoothstep**.

Distances

> **calc.arclen, calc.eucliddist, calc.sinuosity**.

Miscellaneous:

> **calc.Ai, calc.Bi, calc.bernoulli, calc.dilog, calc.euler, calc.gaussian, calc.lambda, calc.polylog, calc.Psi, calc.weier, calc.zeta**.

The functions:

`calc.Ai (x)`

The Airy wave function returns both the first independent solution to the differential equation $y''(x) = x*y$ and its first derivative, for any real `x`.

See also: **calc.Bi**.

`calc.aitken (f, x0 [, eps [, iter]])`

The function finds the limit of the sequence $x_{n+1} = $ `f`(x_n) with initial x_0 and tolerance `eps`, with a maximum of `iter` iterations, using Aitken extrapolation. `f` is a univariate function. `eps` by default is **DoubleEps** and `iter` is 20.

It returns either the approximated limit a and the first derivative at a, `f`$'(a)$, if successful, and **undefined** twice otherwise. The third return is the number of iterations taken to compute the result.

Example: **calc.aitken**($<< $ x -> 1/2*(x + 2/x) $>>$, 1) $\sim=$ **sqrt**(2).

See also: **times**.

`calc.arclen (f, a, b)`

The function returns the arc length (curvilinear length) of a function `f` in one real between the points `a` and `b`.

The function is implemented in Agena and included in the lib/calc.agn file.

See also: **calc.eucliddist**, **calc.sinuosity**.

`calc.bernoulli (n [, eps])`

Computes the `n`-th Bernoulli number Bn and returns a number. `n` should be a non-negative integer. `eps` is an internal bailout value and by default equals **DoubleEps**.

See also: **calc.euler**.

`calc.Bi (x)`

The Airy wave function returns both the second independent solution to the differential equation y"(x) = x*y and its first derivative, for any real x.

See also: **calc.Ai**.

`calc.Ci (x)`

Computes the cosine integral and returns it as a number. `x` must be a number.

See also: **calc.Si**, **calc.Chi**, **calc.Shi**, **calc.Ssi**.

`calc.cheby (f, a, b, n [, ···] [, option])`

Returns a function computing the Chebyshev interpolant for a given point. `f` is the univariate or multivariate function to be interpolated, `a` and `b` represent the domain of the definition, `n` is the order of the interpolant. As a rule of thumb, the wider the domain, the larger `n` should be. If `f` has more than one argument, then all arguments *except the first* are passed right after `n`.

You may optionally pass the `deriv`=k option as the very last argument to compute either the first (k=1), second (k=2), third (k=3), fourth (k=4) or fifth (k=5) derivative, where k defaults to 0, i.e. the function itself.

Using this function may speed up numeric computations significantly if the expression to be evaluated consists of many subexpressions - and if accuracy is not of primary concern. When computing derivatives, it is 10 times faster than **calc.xpdiff** and - depending on the expression type - also more accurate.

Example:

```
> # get derivative of ln(x), i.e. 1/x
> g := calc.cheby(<< x -> ln x >>, 1, 10, 50, deriv = 1);

> g(5):
0.20000000000002
```

See also: **calc.chebycoeffs**, **calc.diff**, **calc.savgol**, **calc.xpdiff**.

`calc.chebycoeffs (f, a, b, n)`

Computes Chebyshev interpolation coefficients used internally by **calc.cheby**. `f` is a univariate function for which coefficients shall be computed, `a` and `b` represent the domain of the definition, `n` is the order of the interpolant.

The return is a table of the Chebyshev coefficients, indexed from 1 to `n`, with key `domain` representing the domain `a`:`b` (a pair). As a rule of thumb, the larger the domain, the larger `n` should be. See also: **calc.savgolcoeffs**.

`calc.chebyt (n, x)`

Computes the `n`-th Chebyshev polynomial of the first kind, evaluated at `x`, with `n` a non-negative integer and `x` a number. The return is equal to **cos**(n***arccos**(x)).

`calc.Chi (x)`

Computes the hyperbolic cosine integral and returns it as a number. `x` must be a number.

See also: **calc.Si**, **calc.Ci**, **calc.Shi**, **calc.Ssi**.

`calc.clampedspline (obj, da:db)`

`calc.clampedspline (obj, da:db, a)`

`calc.clampedspline (obj, da:db, a, coeffs)`

Evaluates the clamped cubic spline for a given table or sequence `obj` of pairs representing the points $x_k:y_k$, at a single value `a` (a number) of the independent variable x.

The boundary conditions are passed as a pair of numbers `da:db`, where `da` is the derivative of the function at the left border, and `db` is the derivative of the function at the right border.

In the first form, returns a univariate function which can be called with a number to obtain the value of the interpolating polynomial. For best performance, use this first form.

In the second form, the function computes the coefficients of the linear, quadratic, and cubic terms itself in each call.

In the third form, the function expects the coefficients `coeffs` of the linear, quadratic, and cubic terms as a sequence of three sequences, in this order, and each containing numbers. The fourth argument may be obtained by calling **calc.clampedsplinecoeffs**.

In the second and third form, the function returns the value of the interpolating polynomial, a number, at the specified value `a` of the independent variable x.

In general, the function returns **fail** if the structure contains less than two pairs.

See also: **calc.interp**, **calc.clampedsplinecoeffs**, **calc.nakspline**, **calc.neville**.

`calc.clampedsplinecoeffs (obj, da:db)`

Determines the coefficients for the clamped cubic spline for a given table or sequence `obj` of pairs representing the points $x_k:y_k$. The return can be used to speed up execution of **calc.clampedspline**.

The boundary conditions are passed as a pair of numbers `da:db`, where `da` is the derivative of the function at the left border, and `db` is the derivative of the function at the right border.

The function returns **fail** if the structure less than two pairs.

See also: **calc.clampedspline**.

`calc.curvature (f, x [, ···])`

Determines the curvature of a real univariate or multivariate function at a given point. Curvature in this context is defined as the rate of change of *direction* of a point that moves on a curve. The result is the number $f''(x, \cdots)/(1 + f'(x, \cdots)^2)^{3/2}$.

The function is implemented in Agena and included in the lib/calc.agn file.

See also: **calc.sinuosity**.

`calc.dawson (x)`

Computes Dawson's integral for a number `x`. The return is the number

$$e^{-x^2} \int_0^x e^{t^2} dt.$$

See also: **calc.scaleddawson**, **expx2**.

`calc.diff (f, x [, ···] [, options])`

Computes the value of the first derivative of a function `f` at a point `x`, and also returns the absolute error as a second return.

If the option `deriv`=n is given, where n may be 1, 2, or 3, the n-th derivative is calculated, with n = 1 the default. If n = 0, then the function value at `f(x)` is determined.

If the `eps`=h option is given, the epsilon value h (a positive number preferably close to zero) is used to determine the difference quotient; otherwise it is automatically determined by calling **math.epsilon** with `x`.

If the absolute error is quite large, it may either indicate non-differentiability of `f` at `x`, ···, or that the derivative could not be computed with sufficient precision.

The algorithm is based on Conte and de Boor's `Coefficients of Newton form of polynomial of degree 3`, and computes symmetric difference quotients.

See also: **calc.cheby**, **calc.differ**, **calc.isdiff**, **calc.eulerdiff**, **calc.xpdiff**.

`calc.differ (f, x [, ···] [, options])`

Computes the derivative of a univariate or multivariate function `f` at point `x` (a number). The second, etc. arguments to `f` may be given right after argument `x`.

If the option `deriv`=n is given, where `n` may be 1, 2, or 3, the n-th derivative is calculated, with n = 1 the default. If n = 0, then the function value at f(x) is determined.

If the `eps`=h option is given, the epsilon value h (a positive number preferably close to zero) is used to determine the difference quotient; otherwise it is automatically determined by calling **math.epsilon** for `x`.

This function actually is just a simple wrapper to **calc.eulerdiff** if `deriv`=1 or no `deriv` option has been given, and to **calc.xpdiff** otherwise, thus automatically choosing the best method to compute the derivative.

The function is implemented in Agena and included in the lib/calc.agn file.

calc.dilog (x)

Computes the dilogarithm (Spence's) function for a number x. The return is the number

$$Li_2(x) = \sum_{k=1}^{\infty} \frac{z^k}{k^2}$$

See also: **calc.polylog**.

calc.Ei (x)

Computes the exponential integral

$$Ei(x) = -\int_{-x}^{\infty} \frac{e^{-t}}{t} dt$$

for a number x. The return is a number[20], and **undefined** if $x = 0$.

calc.En (n, x)

Evaluates the exponential integral

$$E_n(x) = -\int_{1}^{\infty} \frac{e^{-xt}}{t^n} dt$$

for non-negative n (an integer) and real x. The return is a number.

calc.eucliddist (f, a, b [, $\cdots$])

Computes the Euclidian distance, i.e. the straight-line distance, of two points (a, $f(a)$) and (b, $f(b)$) on a curve defined by a function f in one real, in the Euclidean plane. a, b must be numbers. If f is multivariate, its second, third, etc. argument are passed after b.

See also: **|-** operator, **hypot**, **calc.sinuosity**.

calc.euler (n [, eps])

Computes the n-th Euler number En and returns a number. n should be a non-negative integer. eps is an internal bailout value and by default equals **DoubleEps**.

See also: **calc.bernoulli**.

---

[20] Please note that for $-5 \leqslant x < 0$, the result is an approximation.

`calc.eulerdiff (f, x [, ···] [, option])`

Computes the first (second, third, see below) derivative of the univariate or multivariate function `f` at real point `x`, a number.

If the option eps=h is given, the epsilon value h (a positive number preferably close to zero) is used internally for the computation, its default is **math.epsilon**(`x`).

The second, etc. arguments to `f` may be given right after argument `x`.

The return is the imaginary part of `f`(`x` + I*h)/h, or **fail** if `f` did not evaluate to the complex plane. The function does not check whether `f` is differentiable at `x`, ···, you may call **calc.isdiff** before.

If the deriv=2 option is passed, then the second derivative is determined, deriv=3 computes the third derivative. The quality of the second and third derivatives are close to, but not as good as, the one of **calc.xpdiff**. (The first derivatives of **eulerdiff** are still better than those of **xpdiff**.)

This function as at least three times faster than **calc.xpdiff**. See also: **calc.diff**.

The idea has been taken from the Euler Math Toolbox, thus its name.

See also: **calc.differ**, **calc.diff**, **calc.xpdiff**.

`calc.fminbr (f, a, b [, tol])`

Estimates the minimum location of a univariate function `f` through one-dimensional search over a given range [`a`, `b`], with `a`, `b` numbers, using Golden section search combined with parabolic interpolation. The acceptable tolerance `tol` defaults to **Eps**. Returns the *abscissa* (x-axis) value where a minimum has been found, a number.

If there are multiple minima in the range, the function returns an arbitrary one. This function is rather basic, see **calc.minimum** and **calc.maximum** for alternatives.

See also: **calc.fmings**.

`calc.fmings (f, a, b [, tol])`

Like **calc.fminbr**, but performs Golden section search only.

Given a function `f` with a single local minimum in the interval [`a`, `b`], with `a`, `b` numbers, returns the *abscissa* value (x-axis) where a minimum has been found. The acceptable tolerance `tol` defaults to **DoubleEps**.

`calc.fprod (f, a, b)`

Computes the product of `f`(a), $\cdots$, `f`(b), with `f` a function, `a` and `b` numbers. If `a > b`, then the result is 1.

See also: **calc.fsum**.

`calc.fresnelc (x)`

Computes the Fresnel integral $C(x) = \int\limits_{0}^{x} \cos(\frac{\pi}{2} t^2)\, dt$ and returns it as a number.

`calc.fresnels (x)`

Computes the Fresnel integral $S(x) = \int\limits_{0}^{x} \sin(\frac{\pi}{2} t^2)\, dt$ and returns it as a number.

`calc.fsum (f, a, b [, ···])`

Computes the sum of `f`(a), $\cdots$, `f`(b), with `f` a function, `a` and `b` numbers. If `f` requires two or more arguments, the second, third, etc. argument must be passed after `b`. If `a > b`, then the result is 0. The function uses Kahan-Babuška round-off error prevention. Examples:

```
> calc.fsum(<< n, x -> (x**n)/fact(n) >>, 0, 100, 1):
2.718281828459

> calc.fsum(<< x, n -> (x**n)/fact(n) >>, 0, 100, 1):
5050
```

See also: **qsumup**, **sumup**, **calc.fprod**, **stats.cumsum**, **stats.fsum**.

`calc.gauleg (f, a , b [, n, [, eps]])`

Performs Gauss-Legendre integration of a real univariate function `f` over the interval `a .. b` with `n` sample points over the whole range, and precision `eps`. `n` defaults to 50 and `eps` to **DoubleEps**. It is seven times faster than **calc.integ** with similar precision.

`calc.gaussian (x [, a [, b [, c]]])`

Computes the Gaussian function $\frac{a}{2c^2}\, e^{-(x-b)^{\wedge}2}$ at a real or complex point `x`, with `a`, `b`, `c` being (real) numbers. By default, $a = 1$, $b = 0$, $c = \frac{1}{\sqrt{2}}$. The return depends on the type of `x`.

See also: **expx2**.

`calc.gd (x)`

Computes Gudermannian function for any number or complex number `x`, i.e.

$$gd(x) = \int_{t=0}^{x} \frac{1}{\cosh(t)}\, dt = \arctan(\sinh(x)).$$

The type of return depends on x.

Wikipedia: `The Gudermann function relates circular functions and hyperbolic functions without explicitly using complex numbers.` The function is written in the Agena language.

See also: **calc.logistic**, **calc.sigmoid**.

`calc.gtrap (f, a, b [, eps])`

Integrates the function `f` on the interval [`a`, `b`] using a bisection method based on the trapezoid rule and returns a number. By default the function quits after an accuracy of `eps` = **Eps** has been reached. You may pass another numeric value for `eps` if necessary.

See also: **calc.intde**, **calc.intdei**, **calc.intdeo**, **calc.integ**, **calc.simaptive**.

`calc.ibeta (x, a, b)`

Evaluates the incomplete beta integral defined by

$$\frac{\Gamma(a+b)}{\Gamma(a)\Gamma(b)} \int_{0}^{x} t^{a-1}(1-t)^{b-1}\, dt$$

from 0 to x. Both `a` and `x` must be positive numbers. See also: **calc.invibeta**.

`calc.igamma (x, a)`

Evaluates the incomplete gamma integral defined by

$$\frac{1}{\Gamma(a)} \int_{0}^{x} e^{-t}\, t^{a-1}\, dt$$

Both `a` and `x` must be positive numbers. See also: **calc.igammac**.

`calc.igammac (x, a)`

Evaluates the complemented incomplete gamma integral defined by

$$\frac{1}{\Gamma(a)} \int_{x}^{\infty} e^{-t} \, t^{a-1} \, dt$$

Both `a` and `x` must be positive numbers. See also: **calc.igamma**.

`calc.intde (f, a, b [, eps])`

Integrates the function `f` on the interval [`a`, `b`], with `a` and `b` numbers, using Double Exponential (DE) Transformation, also known as Tanh-sinh quadrature.

`f` needs to be analytic over [`a`, `b`]. `eps` is the relative error requested excluding cancellation of significant digits, and by default is equal to 1e-15. Specifically, `eps` means: (absolute error) / ($\int_{a}^{b}$ f(x)|dx).

The return is 1) the approximation to the integral, or **fail** if evaluation failed, and 2) an estimate err of the absolute error, where

- err $\geq$ 0: normal termination,
- err $<$ 0: abnormal termination, i.e. an convergent error has been detected: 1) f(x) or $\frac{d}{dx}^{n}$ f(x) has discontinuous points or sharp peaks over [`a`, `b`] (you must divide the interval [`a`, `b`] at these points). 2) The relative error of f(x) is greater than `eps`. 3) f(x) has an oscillatory factor and the frequency of the oscillation is very high.

This function is four times faster than **calc.gtrap** and also much more accurate. It can be applied on any polynomial, exponential or trigonometric function, logarithm, power function, and most special functions.

See also: **calc.gauleg**, **calc.gtrap**, **calc.intdei**, **calc.intdeo**, **calc.integ**, **calc.simaptive**.

`calc.intdei (f, a, [, eps])`

Integrates the non-oscillatory function `f` on the interval [`a`, ∞], with `a` a number, using Double Exponential (DE) Transformation, also known as Tanh-sinh quadrature.

`f` needs to be analytic over [`a`, ∞]. `eps` is the relative error requested excluding cancellation of significant digits, and by default is equal to 1e-15. Specifically, `eps` means: (absolute error) / ($\int_{a}^{b}$ f(x)|dx).

The return is either the approximation to the integral, or **fail** if evaluation failed, and an estimate err of the absolute error. For further information see calc.intde.

See also: **calc.gtrap**, **calc.intde**, **calc.integ**, **calc.simaptive**.

`calc.intdeo (f, a, [, omega [, eps])`

Integrates the oscillatory function `f` on the interval [`a`, ∞], with `a` a number, using Double Exponential (DE) Transformation, also known as Tanh-sinh quadrature.

`f` needs to be analytic over [`a`, ∞]. `omega` is the oscillatory factor of `f` and by default is 1. `eps` is the relative error requested excluding cancellation of significant digits, and by default is equal to 1e-15. Specifically, `eps` means: (absolute error)/($\int_{a}^{b}$ f(x)|dx).

The return is either the approximation to the integral, or **fail** if evaluation failed, and an estimate err of the absolute error. For further information see calc.intde.

See also: **calc.gtrap**, **calc.intde**, **calc.intdei**, **calc.integ**, **calc.simaptive**.

`calc.integ (f, a, b [, omega [, eps [, n]]])`

This function is a wrapper around **calc.intde**, **calc.intdei**, and **calc.intdeo**. If `eps` is not given, it is 1e-15 by default. If `omega` is not given, it is 1. The return is the integral value and the error margin, both are numbers.

If `b` is not **infinity**, the function calls **calc.intde** and returns its results.

If `b` is infinity, the function first calls **calc.intdei** and returns its results, if **intdei** does not evaluate to **fail**. Otherwise, **calc.intdeo** is called.

If the function should internally switch to another algorithm in order to compute a result successfully, `n` denotes the number of sample points to evaluate. `n` by default is set to (`b` - `a`)*20, i.e. 20 samples per unit.

The function is implemented in Agena and included in the lib/calc.agn file.

See also: **calc.gauleg**, **calc.gtrap**, **calc.intde**, **calc.intdei**, **calc.intdeo**, **calc.simaptive**.

`calc.interp (obj)`
`calc.interp (obj, a)`
`calc.interp (obj, a, coeffs)`

In the first form, computes a Newton interpolating polynomial and returns it as a univariate function. The interpolation points are passed in a table `obj`, with each point being represented by the pair $x_k : y_k$.

Example:

```
> f := calc.interp([ 0:0, 1:3, 2:1, 3:3 ]);
```

Call f at point 10:

```
> f(10):
885
```

In the second and third form, it evaluates the Newton form of the polynomial which interpolates a given table or sequence `obj` of pairs representing the points $x_k : y_k$, at a single value `a` (a number) of the independent variable.

In the second form, the function computes the coefficients automatically in each call, which slows down this variant.

In the third form, by passing a sequence `coeffs` of coefficients (numbers), the function uses the coefficients passed, avoiding their (re-)computation. The third argument may be obtained by calling **calc.newtoncoeffs**.

Both in second and third form, the function returns the value of the interpolating polynomial, a number, at the specified value `a` of the independent variable. It is advised to use the first form to benefit from maximum speed.

Example:

```
> calc.interp([ 0:0, 1:3, 2:1, 3:3 ], 10):
885
```

See also: **calc.cheby**, **calc.clampedspline**, **calc.linterp**, **calc.nakspline**, **calc.neville**, **calc.newtoncoeffs**, **calc.polyfit**, **calc.savgol**.

calc.invibeta (y, a, b)

Evaluates the inverse of the incomplete beta integral such that

$$y = calc.ibeta(x, a, b).$$

See also: **calc.ibeta**.

calc.iscont (f, x [, ···] [, option])

The function returns **true** if a real function `f` is continuous at the given point, and **false** otherwise. If `f` requires only one argument, `x` is a number. If `f` requires two or more arguments, the second, third, etc. argument of `f` must be passed right after `x`.

If the option `eps`=h is given as the last argument, the epsilon value h (a positive number preferably close to zero) is used for the approximate equality check with the left and right limit; otherwise it is automatically determined by calling **math.eps** with `x` and any option given. See **calc.limit** for an example.

See also: **calc.isdiff**.

calc.isdiff (f, x [, ···] [, option])

The function returns **true** if a real function `f` is differentiable at the given point `x`, of type number, and **false** otherwise. If `f` requires two or more arguments, the second, third, etc. argument (all of type number) - of `f` must be passed right after `x`.

If the option `eps`=h is given as the last argument, the epsilon value h (a positive number preferably close to zero) is used for the approximate equality check with the left and right limit; otherwise it is automatically determined by calling **math.epsilon**(`x`). See **calc.limit** for an example.

A function is differentiable at x, ··· if it is continuous at x and if the left- and right-sided difference quotients are equal.

See also: **calc.diff**, **calc.iscont**, **calc.xpdiff**.

calc.lambda (v, x [, eps])

Computes the Lambda function and its derivative of (positive) integral order `v` for argument `x`:

$$\Lambda_v(x) = 2^v \Gamma(v+1)$$
$$\Lambda'_v(x) = \frac{2v}{x}(\Lambda_{v-1}(x) - \Lambda_v(x))$$

Γ is the Gamma function, and J_v is the Bessel function of the first kind (Agena function **besselj**). The function also returns the actual order processed, which may differ from the input `v`. `eps` is a bailout value and by default is **DoubleEps**.

calc.limit (f, x [, ···] [, options])

The function returns the limit, a number, of a real function `f` at the given point `x` (a number). If the limit does not exist, **undefined** will be returned.

If `f` is multivariate, the second, third, etc. argument of `f` must be passed right after `x`.

Options may be given as the very last arguments, their order does not matter.

If the `eps`=h option is given, the epsilon value h (a positive number preferably close to zero) is used for the approximate equality check of the left and right limit; otherwise it is automatically determined by calling **math.eps** with `x` and any option.

If the `side='left'` option is given, the left-sided limit is determined. If the `side='right'` option is given, the right-sided limit is determined. If the `side='both'` option is given, the left and right-sided limit, in this order, will be returned. If the `side='all'` option is given, the limit, the left-sided, and the right-sided limit will be returned, in this order.

For example, if the function is f(x, y) := |x| + y, with x = 1, y = 3, and eps = 1e-4, the call for the left-sided limit would be:

```
> calc.limit(<< x, y -> abs(x) + y >>, 0, 3, eps = 1e-4, side='left'):
3
```

`calc.linterp (obj)`

Returns a function that conducts a Lagrange interpolation for a given sequence or table `obj` of numeric pairs x:y where x and y denote a point in the plane. It is often said that Lagrange interpolation is suited for theoretical purposes only, since it is rather slow.

See also: **calc.interp**, **calc.polyfit**.

`calc.logistic (x [, max [, k [, x0]]])`

Computes the logistic function, having a characteristic `S`-shaped curve or sigmoid curve, for any number `x`, according to the formula

$$L(x) = \frac{max}{1 + e^{-k\,(x - x_0)}},$$

where `max` is the curve's maximum value, `k` its steepness, and `x0` the x-value of the sigmoid's midpoint. By default, `max` = 1, `k` = 1, and `x0` = 0, computing the sigmoid function. If only `x` is given, it works like **calc.sigmoid**.

The result is a number between - but excluding - 0 and `max`.

See also: **calc.gd**, **calc.logit**, **calc.sigmoid**.

`calc.logit (x)`

Computes the inverse sigmoid or logistic function according to the formula

$$logit(x) = \frac{\ln(x)}{\ln(1+x)} = 2\,arctanh(2x - 1).$$

If $x=0$, the function will return -**infinity**, and if $x=1$ the result will be +**infinity**.

See also: **calc.logistic**, **calc.probit**, **calc.sigmoid**.

`calc.maximum (f, a, b, [step [, eps]])`

Returns all *possible* maximum locations of the univariate function `f` on the interval [`a`, `b`]. The function divides the interval [`a`, `b`] into smaller intervals [`a`, `a+step`], [`a+step`, `a+2*step`], $\cdots$, [`b-step`, `b`], with `step`=0.1 if `step` is not given. It then looks for possible maximum locations `x` in these smaller intervals and checks whether the first derivative of `f` at `x` is 0.

`f` must be differentiable on [`a`, `b`]. The procedure returns two sequences.

The accuracy of the procedure is determined by `eps`, with `eps` = **Eps** as a default. If a possible extreme location `x` matches the condition `f'(x)` = 0 with this accuracy, it is included in the first sequence that the procedure returns. If the test fails and `eps` $\leq$ **Eps**, then an accuracy of 1e-5 is used for a second test. If it succeeds, `x` is included into both the first and the second sequence, indicating to the user that the first test failed.

The function is implemented in Agena and included in the lib/calc.agn file.

See also: **calc.fminbr**, **calc.fmings**, **calc.minimum**.

`calc.minimum (f, a, b, [step [, eps]])`

Returns all *possible* minimum locations of the univariate function `f` on the interval [`a`, `b`]. The function divides the interval [`a`, `b`] into smaller intervals [`a`, `a+step`], [`a+step`, `a+2*step`], $\cdots$, [`b-step`, `b`], with `step`=0.1 if `step` is not given. It then looks for possible minimum locations `x` in these smaller intervals and checks whether the first derivative of `f` at `x` is 0.

`f` must be differentiable on [`a`, `b`]. The procedure returns two sequences.

The accuracy of the procedure is determined by `eps`, with `eps` = **Eps** as a default. If a possible extreme location `x` matches the condition `f'(x)` = 0 with this accuracy, it is included in the first sequence that the procedure returns. If the test fails and `eps` $\leq$ **Eps**, then an accuracy of 1e-5 is used for a second test. If it succeeds, `x` is included into both the first and the second sequence, indicating to the user that the first test failed.

The function is implemented in Agena and included in the lib/calc.agn file.

See also: **calc.fminbr**, **calc.fmings**, **calc.maximum**.

```
calc.nakspline (obj)
```

```
calc.nakspline (obj, a)
```

```
calc.nakspline (obj, a, coeffs)
```

Evaluates the `not-a-knot` cubic spline for a given table or sequence `obj` of pairs representing the points $x_k\!:\!y_k$, at a single value `a` (a number) of the independent variable.

In the first form, returns a univariate function which can be called with a number to obtain the value of the interpolating polynomial. This is the recommended usage due to its run-time behaviour.

In the second form, the function computes the coefficients of the linear, quadratic, and cubic terms itself in each call.

In the third form, the function expects the coefficients `coeffs` of the linear, quadratic, and cubic terms as a sequence of three sequences, in this order, and each containing numbers. The third argument may be obtained by calling **calc.naksplinecoeffs**.

In the second and third form, the function returns the value of the interpolating polynomial, a number, at the specified value `a` of the independent variable.

In general, the function returns **fail** if the structure contains less than four pairs.

See also: **calc.clampedspline**, **calc.interp**, **calc.naksplinecoeffs**, **calc.neville**.


```
calc.naksplinecoeffs (obj)
```

Determines the coefficients for the `not-a-knot` cubic spline for a given table or sequence `obj` of pairs representing the points $x_k\!:\!y_k$. The return can be used to speed up execution of **calc.nakspline**.

The function returns **fail** if the structure contains less than four pairs.

See also: **calc.nakspline**.


```
calc.neville (obj)
```

```
calc.neville (obj, a)
```

In the first form, returns a function that conducts an Aitken-Neville interpolation for a given sequence or table `obj` of numeric pairs $x_k\!:\!y_k$ where x_k and y_k denote a point in the plane.

In the second form, evaluates the polynomial which interpolates a given sequence or table `obj` of points represented by pairs of the form $x_k:y_k$ at a single value `a` (a number) of the independent variable, using Aitken-Neville interpolation, and returns a number.

Example:

```
> calc.neville([1:1, 2:2, 3:3], 2):
2
```

See also: **calc.clampedspline**, **calc.interp**, **calc.nakspline**.

`calc.newtoncoeffs (obj)`

Returns a sequence of the coefficients of type number of the Newton form of the polynomial which interpolates a given table or sequence `obj` of pairs representing the points $x_k:y_k$. The return can be used to speed up execution of **calc.interp**.

See also: **calc.interp**.

`calc.polyfit (obj, n)`

For an `n`-th-degree polynomial of a sample of Cartesian pairs $x_k:y_k$, returns a sequence of coefficients of descending degree, using polynomial regression. x_k, y_k are numbers and degree `n` must be a positive integer. Example:

```
> coeffs := calc.polyfit(seq( 1:0, 2:3, 3:1 ), 2):
seq(-2.5, 10.5, -8.0000000000001)
```

The return may be passed to **calc.polygen** to generate a polynomial function, e.g.:

```
> p := calc.polygen(coeffs);
```

There is no limit on the degree, but a degree of 7 or more is not regarded appropriate.

The function tries to reproduce polynomial trend lines known from spreadsheet applications and internally uses Kahan-Ozawa-Summation for better accuracy.

See also: **calc.interp**, **calc.linterp**, **calc.polygen**.

`calc.polygen (c_n, c_{n-1}, ···, c_2, c_1)`
`calc.polygen (obj)`

Creates a polynomial $p(x) = c_n * x^{n-1} + c_{n-1} * x^{n-2} + ··· + c_2 * x + c_1$ from the coefficients c_n, c_{n-1}, ···, c_2, c_1 and returns it as a new function $<< x -> p(x) >>$, where x and the return p(x) represent numbers. You may alternatively pass the coefficients in a table, sequence or register `obj`.

The function internally uses 80-bit precision floats.

Example: The Taylor series expansion of the sine function, with order 8, is:

$$\sin(x) = x - \frac{1}{3!}x^3 + \frac{1}{5!}x^5 - \frac{1}{7!}x^7 + \ldots$$

So the coefficients in *descending* [sic !] order are:

```
> coeffs := [1, 0, -1/fact(3), 0, +1/fact(5), 0, -1/fact(7)];
```

This generates the function:

```
> taylorsine := calc.polygen(coeffs);
```

```
> taylorsine(1), sin(1):
0.84146825396825  0.8414709848079
```

See also: **calc.polyfit**.

calc.polylog (n, x)

Returns the polylogarithm of order `n` (an integer greater or equals -1) at a real point `x`. The return is a number, or **fail** if `n` < -1 for this situation is not implemented. The polylogarithm of order `n` is defined by the series:

$$\mathrm{Li}_n(x) = \sum_{k=1}^{\infty} \frac{x^k}{k^n}$$

See also: **calc.dilog**.

calc.probit (x)

Computes the inverse of the cumulative distribution function of the standard normal distribution:

$$\mathrm{probit}(x) = \sqrt{2}\, \mathrm{erf}^{-1}(2x-1).$$

If $x=0$, the function will return -**infinity**, and if $x=1$ the result will be +**infinity**.

See also: **calc.logit**,**inverf**.

calc.Psi (x)

Computes the Psi (digamma) function, the logarithmic derivative of the gamma function, for a number x. The return is the number **calc.xpdiff**(**lngamma**(x), x).

See also: **gamma**, **lngamma**.

`calc.regulafalsi (f, a, b [, eps])`

Tries to determine the root of the univariate function `f` in the borders `a` and `b` and returns it as a number if successful, and **null** otherwise.

The precision `eps` by default is **Eps**.

See also: **calc.zeroin**, **calc.zeros**.

`calc.savgol (f [, options])`

Computes a Savitzky–Golay filter for the univariate function `f` to `smooth` its data by returning a factory interpolating `f` at a given point x_0. preventing large oscillations between sample points.

It fits successive subsets of neighbouring data with a low-degree polynomial using the linear least-square method. By default, 15 equally-spaced points to the left of x_0 and 15 equally-spaced points to the right of x_0 are examined.

You can change this `window` by passing another odd value with the `'points'` option. All adjacent points are separated by distance eps which is 1e-5 by default. You can change the distance with the `'eps'` option, e.g. eps=0.1 which unless you want to compute a derivative, see below, might be a much more useful value.

Alternatively it can also compute derivatives of any degree `n` by passing the option deriv=`n`. The larger the degree `n` of the derivative, however, the less accurate the results will become.

The degree `d` of the smoothing least-square polynomial is 3 by default and can be changed by the degree=`d` option. Recommended degrees are `d` = 2 or 4, with `d` not exceeding 6.

The function automatically determines the most suitable settings for the window and the spacing eps of its points, but you can switch this off by passing the `adaptive=false` option (default is `adaptive=true`).

Example: Compute the first, second and third derivative of sin(x) and evaluate at x= $\frac{\pi}{2}$:

```
> import calc

> f := << x -> sin x >>

> f' := calc.savgol(f, deriv = 1)

> f'' := calc.savgol(f, deriv = 2)

> f''' := calc.savgol(f, deriv = 3)
```

```
> f(Pi/2), f'(Pi/2), f''(Pi/2), f'''(Pi/2):
1   -6.3527471044073e-017   -1.0000000010326   4.3874822912549e-007
```

See also: **calc.cheby**, **calc.interp**, **calc.savgolcoeffs**.

`calc.savgolcoeffs (nleft, nright, deriv, polydeg)`

Returns the normalised Savitzky-Golay filter coefficients as a register. `nleft` is the number of leftward observations to be examined, while `nright` is the number of rightward ones.

`deriv` is the order of the derivative desired (0 for the smoothed function, 1 for the first derivative, asf.).

`polydeg` is the order of the smoothing polynomial, with 2 or 4 being recommended values, but not exceeding 6.

See also: **calc.chebycoeffs**, **calc.savgol**.

`calc.scaleddawson (x)`

Implements the Scaled Dawson Integral w_im(x) = 2***calc.dawson**(x)/**sqrt(Pi)** for real x.

See also: **calc.dawson**, **calc.w**.

`calc.sections (f, a, b, step)`

Returns all intervals where a function has a change in sign. `f` must be a function, `a` the left border of the main interval, `b` its right border, and `step` the step size. The return is a sequence of pairs denoting the found subintervals. The function uses Adapted Neumaier summation to prevent round-off errors, the same used by numeric **for** loops with fractional step sizes.

See also: **calc.zeros**.

`calc.Shi (x)`

Computes the hyperbolic sine integral and returns it as the number

$$\text{Shi}(x) = \int_0^x \frac{\sinh t}{t}\, dt$$

x must be a number.

See also: **calc.Ci**, **calc.Chi**, **calc.Si**, **calc.Ssi**.

`calc.Si (x)`

Computes the sine integral

$$Si(x) = \int_{t=0}^{x} \frac{\sin(t)}{t}\, dt = \int_{t=0}^{x} \text{sinc}(t)\, dt$$

and returns it as a number. `x` must be a number.

See also: **calc.Ci, calc.Chi, calc.Shi, calc.Ssi, sinc**.

`calc.sigmoid (x)`

Computes the sigmoid, i.e. standard logistic, function,

$$S(x) = \frac{1}{1 + e^{-x}}$$

having a characteristic `S`-shaped curve or sigmoid curve, for any number `x`. The result is a number between - but excluding - 0 and 1.

See also: **calc.gd, calc.logistic, calc.logit**.

`calc.simaptive (f, a, b [, h_min [, eps]])`

Integrates the function `f` on the interval [`a`, `b`] using Simpson-Simpson Adaptive Quadrature and returns a number. The function returns **fail**, if no suitable subinterval of length greater than `min_h` could be found for which the estimated error falls below `eps`.

The function is thrice as fast as **calc.integ**, but is not suited with singularities at or within the borders.

By default, `h_min` is 1e-7, and `eps` is **Eps**/2, where **Eps** is the global system variable **Eps**.

See also: **calc.gauleg, calc.gtrap, calc.intde, calc.intdei, calc.intdeo, calc.integ**.

`calc.sinuosity (f, a, b)`

Computes the ratio of the curvilinear length (along the curve) and the Euclidean distance (straight line) between the end points `a` and `b`, of the curve defined by a function `f` in one real. `a, b` must be numbers.

sinuosity does not check whether `f` has an inflection point in the given range [`a,b`]. The function is implemented in Agena and included in the lib/calc.agn file.

See also: **calc.arclen**, **calc.curvature**, **calc.eucliddist**.

`calc.smoothstep (x, n)`

`calc.smoothstep (x, 'perlin')`

In the first form, the function receives a non-negative integer `n` and any number `x` and returns 0 if `x` < 0, 1 if `x` > 1, and smoothly interpolates between 0 and 1, using an (2*`n`+1)'th-degree Hermite polynomial otherwise.

The slope of the smoothstep function is zero at both edges, so the result is differentiable over the whole real domain.

Wikipedia: `Smoothstep is a family of sigmoid-like interpolation and clamping functions commonly used in computer graphics and video game engines`, for example to naturally accelerate or decelerate an object.

In the second form, if the string `'perlin'` is passed, the function computes the `smootherstep` polynomial $6x^5 - 15x^4 + 10x^3$ for 0 < `x` < 1, which has zero 1st- and 2nd-order derivatives at `x` = 0 and `x` = 1, as suggested by computer scientist Prof. Kenneth Perlin. If given, `n` may be any integer as it is not evaluated in this case.

See also: **heaviside**, **math.clip**, **math.rectangular**, **math.unitise**.

`calc.softsign (x)`

Computes the Softsign function $\dfrac{x}{1 + |x|}$.

`calc.Ssi (x)`

Computes the shifted sine integral and returns it as a number. `x` must be a number.

See also: **calc.Ci**, **calc.Chi**, **calc.Shi**, **calc.Si**.

`calc.variance (f, a, b [, relative [, n [, eps]]])`

Returns a positive integer that indicates whether a function `f` in one real changes slowly or rapidly on the given interval [`a`, `b`], with `a`, `b` numbers. The larger the result, the larger is its rate of change.

By default,
- the result is `relative`, i.e. given per unit on the abscissa (**true** for `relative`, **false** for absolute),
- there are `n` =10 sample points per unit, and
- the bail-out value `eps` - a positive value close to zero - is **Eps**.

Internally, the function uses adaptive integration with trapezoidal rule and counts the number of trapezoids evaluated. Note that the results are estimates.

`calc.w (z [, eps])`

Implements the scaled complex complementary error function w(z) = **exp**(-z^2)***erfc**(-**I***z) (Faddeeva function) for number or complex number z. The return is a complex value. By default, the precision `eps` is **DoubleEps**, and can be any other non-negative number.

See also: **calc.scaleddawson**.

`calc.weier (x, a, b [, eps])`

Implements the Weierstraß function for the given number x and parameters a, b (also numbers), with $0 < x < 1$ and $ab \geq 1$, b an odd positive integer - a function that is continuous but non-differentiable everywhere:

$$\sum_{n=0}^{\infty} a^n \cos(b^n x)$$

The precision is given by its fourth optional argument, `eps`, which is **Eps** by default.

The function internally uses Kahan-Ozawa-Summation for better accuracy.

`calc.xpdiff (f, x [, ···] [, options])`

Like **calc.diff**, but uses Richardson's extrapolation method to compute symmetric difference quotients. `f` is a univariate or multivariate function to be inspected at point `x` (a number). The second, etc. arguments to `f` may be given right after argument `x`.

The return of the procedure is the derivative of `f` at `x`, ···, - a number - and the absolute error. If the absolute error is quite large, it may indicate non-differentiability of `f` at `x`, ···. If the function could not determine a result if `x` is near an undefined domain, it automatically calls **calc.diff**, which is more robust in this situation but returns less precise results.

If the option `deriv`=n is given, where n may be 1, 2, or 3, the n-th derivative is calculated, with n = 1 the default. If n = 0, then the function value at `f`(x) is determined.

If the option `eps`=h is given, the epsilon value h (a positive number preferably close to zero) is used for the relative error check. If the option `delta`=g is given, the delta value g (a positive number preferably close to zero) is used for the absolute error check, otherwise they both default to **math.epsilon**(x).

xpdiff produces better results with powers and trigonometric functions than **calc.diff**.

See also: **calc.diff**, **calc.differ**, **calc.eulerdiff**, **calc.isdiff**.

`calc.zeroin (f, a, b [, eps])`

Determines the root of the univariate function `f` in the borders `a` and `b` and returns a number if successful, and **null** otherwise.

The precision `eps` by default is **Eps**.

In general, the function will even return accurate results where **calc.regulafalsi** fails to do so - or even cannot find a root at all -, but the runtime behaviour compared to **calc.regulafalsi** depends on the following conditions:

1. the interval should not be too far from the origin,
2. the width of the interval should not be too small.

If both conditions are met, then the function can be faster than **calc.regulafalsi**.

The algorithm uses bisection combined with linear or quadric inverse interpolation, followed by applying Regula Falsi to the estimate done by the previous actions.

See also: **calc.regulafalsi**, **calc.zeros**.

`calc.zeros (f, a, b, [step [, eps]])`

Returns all roots of a function `f` in one variable on the interval [`a`, `b`] in a sequence. If it could not find a root, it returns **null**.

The function divides the interval [`a`, `b`] into smaller intervals [`a`, `a+step`], [`a+step`, `a+2*step`], $\cdots$, [`b-step`, `b`], with `step`=0.1 if `step` is not given. It then looks for changes in sign in these smaller intervals and if it finds them, determines the roots using a modified regula falsi method.

The accuracy of the regula falsi method is determined by `eps`, with `eps` = **Eps** as a default. `f` must be differentiable on [`a`, `b`].

The function is implemented in Agena and included in the lib/calc.agn file.

See also: **calc.regulafalsi**, **calc.sections**, **calc.zeroin**.

`calc.zeta (x)`

Computes the Riemann Zeta function for real `x` > 1 and returns the number:

$$\sum_{k=2}^{\infty} k^{-x} + 1$$

11.13 linalg - Linear Algebra Package

This package provides basic functions for Linear Algebra. As a *plus* package, it is not part of the standard distribution and must be activated with the **import** statement, e.g. `import linalg`.

There are two constructors available to define vectors and matrices, **linalg.vector** and **linalg.matrix**. Except of these two procedures, the package functions assume that the geometric objects passed have been created with the above mentioned constructors.

The package includes a metatable **linalg.vmt** defined in the lib/linalg.agn file with metamethods for vector addition, vector subtraction, and scalar vector multiplication. Further functions are provided to compute the length of a vector with the **abs** operator and to apply unary minus to a vector.

The table **linalg.mmt** defines metamethods for matrix addition, subtraction and multiplication with a scalar. It is assigned via the lib/linalg.agn file, as well.

The **vector** function allows to define sparse vectors, i.e. if the component n of a vector v has not been physically set, and if v[n] is called, the return is 0 and not **null**.

The dimension of the vector and the dimensions of the matrix are indexed with the `'dim'` key of the respective object. You should not change this setting to avoid errors. Existing vector and matrix values can be overwritten but you should take care to save the correct new values.

> Equality checks of vectors or matrices should always be conducted with the strict equality operator == or the ~= approximate equality operator instead of the Cantor-like = equality operator[21]. For inequality use the **not** operator combined with == or ~=.

A sample session:

```
> import linalg alias
```

Define two vectors in two fashions: In the simple form, just pass all components explicitly:

---

[21] The = operator just checks whether an element in one structure is residing at any position in the other structure, whereas the == and ~= operators check elements place-by-place. Developers who would like to extend the **linalg** package may also have a look at the __eeq and __aeq metamethod. to influence the behaviour of the == and ~= operators, respectively.

```
> a := vector(1, 2, 3):
[ 1, 2, 3 ]
```

In a more elaborate form, indicate the dimension of the vector to be created and only pass the vector components that are not zero in a table:

```
> b := vector(3, [1~2]):
[ 2, 0, 0 ]
```

Check whether a and b are parallel and have the same direction:

```
> abs(a+b) = abs(a) + abs(b):
false
```

Addition:

```
> a + b:
[ 3, 2, 3 ]
```

Subtraction:

```
> a - b:
[ -1, 2, 3 ]
```

Scalar multiplication:

```
> 2 * a:
[ 2, 4, 6 ]

> crossprod(a, b):
[ 0, 6, -4 ]
```

Find the vector x which satisfies the matrix equation A x = b. In this example, we will solve the equation $\begin{bmatrix} 1 & 2 & -4 \\ 2 & 1 & 3 \\ -3 & 1 & 6 \end{bmatrix} * x = \begin{bmatrix} -6 \\ 5 \\ -2 \end{bmatrix}$. The **linalg.matrix** constructor expects row vectors.

```
> A := matrix([1, 2, -4], [2, 1, 3], [-3, 1, 6]):
[  1, 2, -4 ]
[  2, 1,  3 ]
[ -3, 1,  6 ]

> b := vector(-6, 5, -2):
[ -6, 5, -2 ]

> backsubs(A, b):
[ 2, -2, 1 ]
```

The **linalg** operators and functions are:

s1 + s2

Adds two vectors or matrices `s1`, `s2`. The return is a new vector or matrix. This operation is done by applying the `__add` metamethod.

s1 - s2

Subtracts two vectors or matrices `s1`, `s2`. The return is a new vector or matrix. This operation is done by applying the `__sub` metamethod.

k * s
s * k
m1 * m2

Multiplies a number `k` with each element in vector or matrix `s`, or multiplies the matrix `m1` with matrix `m2`. The return is a new vector or matrix. This operation is done by applying the `__mul` metamethod.

s / k

Divides each element in the vector s by the number `k` The return is a new vector. This operation is done by applying the `__div` metamethod.

abs (v)

Determines the length of vector `v`. This operation is done by applying the `__abs` metamethod to `v`.

qsumup (v)

Raises all elements in vector `v` to the power of 2. The return is the sum of these powers, i.e. a number. This operation is done by applying the `__qsumup` metamethod to `v`.

linalg.add (v, w)

Determines the vector sum of vector `v` and vector `w`. The return is a vector.

See also: **linalg.sub**.

linalg.augment (···)

Joins two or more matrices or vectors together horizontally. Vectors are supposed to be column vectors. The matrices and vectors must have the same number of rows.

The return is a new matrix.

See also: **linalg.stack**.

`linalg.backsub (A)`
`linalg.backsub (A, v)`

Performs backward substitution on a system of linear equations.

In the first form, `A` must be an augmented m x n lower triangular matrix with m+1 = n. In the second form, `A` is an lower triangular square matrix and `v` a right-hand side vector.

The return is the solution vector.

The function issues an error if `A` is not upper triangular. You may change the tolerance to detect `zeros` by setting the global system variable **Eps** to another value.

See also: **linalg.gsolve**, **linalg.rref**.

`linalg.backsubs (A, b)`

The function has been deprecated. Please use **linalg.gsolve** instead.

`linalg.checkmatrix (A [, B, ···] [, true])`

Issues an error if at least one of its arguments is not a matrix. If the last argument is **true**, then the matrix dimensions are returned as a pair, else the function returns nothing.

Contrary to **linalg.checkvector**, the dimensions will not be checked if you pass more than one matrix.

`linalg.checksquare (A)`

Issues an error if `A` is not a square matrix. It returns nothing. See **linalg.issquare** for information on how this check is being done.

`linalg.checkvector (v [, w, ···])`

Issues an error if at least one of its arguments is not a vector. In case of two or more vectors it also checks their dimensions and returns an error if they are different.

If everything goes fine, the function will return the dimensions of all vectors passed.

See **linalg.isvector** for information on how the check is being done.

`linalg.coldim (A [, ···])`

Determines the column dimension of the matrix `A`. The return is a number.

If you pass more than one argument, then a time-consuming check whether `A` is a matrix, is skipped.

A more direct way of determining the column dimension is `right(A.dim)`.

See also: **linalg.rowdim**.

linalg.column (A, n)

Returns the n-th column of the matrix or row vector `A` as a new vector.

See also: **columns**, **linalg.submatrix**.

linalg.crossprod (v, w)

Computes the cross-product of two vectors `v`, `w` of dimension 3. The return is a vector.

linalg.det (A)

Computes the determinant of the square matrix `A`. The return is a number. With singular matrices, it returns 0.

linalg.diagonal (v)

Creates a square matrix `A` with all vector components in `v` put on the main diagonal. The first element in `v` is assigned `A[1][1]`, the second element in `v` is assigned `A[2][2]`, etc. Thus the result is a dim(`v`) x dim(`v`)-matrix.

See also: **linalg.getdiagonal**.

linalg.dim (A)

Determines the dimension of a matrix or a vector `A`. If `A` is a matrix, the result is a pair with the left-hand side representing the number of rows and the right-hand side representing the number of columns. If `A` is a vector, the size of the vector is determined.

linalg.dotprod (v, w)

Computes the vector dot product of two vectors `v`, `w` of same dimension. The vectors must consist of Agena numbers. The return is a number.

```
linalg.forsub (A)
linalg.forsub (A, v)
```
Performs forward substitution on a system of linear equations.

In the first form, `A` must be an augmented m x n upper triangular matrix with m+1 = n. In the second form, `A` is an upper triangular square matrix and `v` a right-hand side vector.

The return is the solution vector.

The function issues an error if `A` is not upper triangular. You may change the tolerance to detect `zeros` by setting the global system variable **Eps** to another value.

See also: **linalg.backsub**, **linalg.rref**.


```
linalg.getdiagonal (A)
```
Returns the diagonal of the square matrix `A` as a vector.

See also: **linalg.diagonal**.


```
linalg.gsolve (A [, true])
linalg.gsolve (A, v [, true])
```
Performs Gaussian elimination on a system of linear equations.

In the first form, `A` must be an augmented m x n matrix with m+1 = n. In the second form, `A` is a square matrix and `v` a right-hand side vector.

The return is the solution vector. It returns **infinity** if an infinite number of solutions has been found, and **undefined** if no solutions exists. It returns **fail** if it could not determine whether no or an infinite number of solutions exist.

If the Boolean value **true** is given as the last argument, the reduced linear system is also returned as an (augmented) upper triangular matrix.

See also: **linalg.backsub**, **linalg.forsub**, **linalg.rref**.


```
linalg.hilbert (n [, x])
```
Creates a generalised `n` x `n` Hilbert matrix H, with H[i, j] := 1/(i+j-x). If `x` is not specified, then `x` is 1. (`n` and `x` must be numbers.)

`linalg.identity (n)`

Creates an identity matrix of dimension `n` with all components on the main diagonal set to 1 and all other components set to 0.

`linalg.inverse (A)`

Returns the inverse of the square matrix `A`.

`linalg.isallones (A)`

Checks whether the vector or matrix `A` contains only ones and returns **true** or **false**.

See also: **linalg.iszero**.

`linalg.isantisymmetric (A)`

Checks whether the matrix `A` is an antisymmetric matrix. If so, it returns **true** and **false** otherwise.

`linalg.isdiagonal (A)`

Checks whether the matrix `A` is a diagonal matrix. If so, it returns **true** and **false** otherwise.

`linalg.isidentity (A)`

Checks whether the matrix `A` is an identity matrix. If so, it returns **true** and **false** otherwise.

`linalg.ismatrix (A)`

Returns **true** if `A` is a matrix, and **false** otherwise. To avoid costly checks of the passed object, the function only checks whether `A` is a sequence with the user-defined type 'matrix'.

`linalg.issquare (A)`

Returns **true** if `A` is a square matrix, i.e. a matrix with equal column and row dimensions, and **false** otherwise.

`linalg.issymmetric (A)`

Checks whether the matrix `A` is a symmetric matrix. If so, it returns **true** and **false** otherwise.

`linalg.isvector (A)`

Returns **true** if `A` is a vector, and **false** otherwise. To avoid costly checks of the passed object, the function only checks whether `A` is a sequence with the user-defined type 'vector'.

`linalg.iszero (A)`

Checks whether the vector or matrix `A` contains only zeros and returns **true** or **false**.

See also: **linalg.isallones**.

`linalg.ludecomp (A [, n])`

Computes the LU decomposition of the square, non-singular matrix `A` of order `n`. If `n` is missing, it is determined automatically, i.e. `n := left(A.dim)`.

The return is the resulting matrix, the permutation vector as a vector, and a number where this number is either 1 for an even number of row interchanges done during the computation, or -1 if the number of row interchanges was odd. If the matrix is singular, an error will be issued.

`linalg.matrix (obj₁, obj₂, ···, objₙ)`

`linalg.matrix (m, n [, lv])`

In the first form, creates a matrix from the given structures obj_k. The structures are considered to be row vectors. Valid structures are vectors created with **linalg.vector**, tables or sequences.

In the second form, with m and n integers, creates a `m` x `n` matrix and optionally fills it row by row with the elements in the table or sequence `lv`. `lv` must not include structures. If `lv` is not given, the matrix is filled with zeros.

The return is a table of the user-defined type 'matrix' and a metatable **linalg.mmt** assigned to the matrix. The table key `'dim'` contains a pair with the dimensions of the matrix: the left-hand side specifies the number of rows, the right-hand side the number of columns.

See also: **linalg.vector**, **utils.readcsv**.

`linalg.maeq (A, B)`

This function checks matrix `A` and matrix `B` for approximate equality. The return is either **true** or **false**. The function uses Donald Knuth's approximation method to compare matrix elements (see the **approx** function for information on how this works).

You can change the accuracy threshold epsilon with the **environ.kernel/eps** function.

See also: ~= and ~<> metamethods, **approx**, **linalg.meeq**, **linalg.vaeq**.

`linalg.meeq (A, B)`

This function checks matrix `A` and matrix `B` for strict equality. The return is either **true** or **false**.

See also: == metamethod, **linalg.maeq**, **linalg.veeq**.

`linalg.mmap (f, A [, ···])`

This function maps a function `f` to all the components in the matrix `A` and returns a new matrix. The function must return only one value. See **linalg.vmap** for further information.

`linalg.mmul (A, B)`

This function multiplies an m x n matrix `A` with an n x p matrix `B`. The return is an m x p matrix. See also: ***** metamethod.

`linalg.mulrow (A, i, s)`

Multiplies each element of row `i` in matrix `A` with the scalar `s` and returns a new matrix.

See also: **linalg.swapcol**, **linalg.swaprow**, **linalg.mulrowadd**.

`linalg.mulrowadd (A, i, j, s)`

Returns a copy of matrix `A` with each element in row `j` exchanged by the sum of this element and the respective element in row `i` multiplied by the number `s`.

See also: **linalg.swapcol**, **linalg.swaprow**, **linalg.mulrowadd**.

`linalg.mzip (f, A, B [, ···])`

This function zips together two matrices `A`, `B` by applying the function `f` to each of its respective components. The result is a new matrix m where each element m[i, j] is determined by m[i, j] := `f`(`A`[i, j], `B`[i, j]). If the `f` has more than two arguments, then its third to last argument must be given right after `B`.

`A` and `B` must have the same dimension.

See also: **linalg.vzip**, **linalg.mmap**, **linalg.mzip**.

`linalg.norm (A)`

`linalg.norm (v [, n])`

The function returns the norm of a matrix or vector.

In the first form, the function returns the infinity norm of a matrix `A`. It is the maximum row sum, where the row sum is the sum of the absolute values of the elements in a given row.

In the second form, it returns the n-norm of a vector `v`, where `n` is a positive integer. (The n-norm of a vector is the n-th root of the sum of the magnitudes (absolute values) of each element in v raised to the n-th power.) If `n` is **infinity**, the return is the infinity norm, i.e. the maximum magnitude of all elements `v`.

`linalg.reshape (A, m [, n])`

Returns an `m` x `n` matrix whose elements are taken from the matrix `A`. The elements of the matrix are accessed in column-major order. If `n` is omitted, it is set to 1.

Example:

```
> a := linalg.matrix(3, 2, [1, 2, 3, 4, 5, 6]):
[ 1, 2 ]
[ 3, 4 ]
[ 5, 6 ]

> reshape(a, 2, 3):
[ 1, 3, 5 ]
[ 2, 4, 6 ]
```

`linalg.rowdim (A [, ···])`

Determines the row dimension of the matrix `A`. The return is a number.

If you pass more than one argument, then a time-consuming check whether `A` is a matrix, is skipped.

A more direct way of determining the column dimension is `left(A.dim)`.

See also: **linalg.coldim**.

`linalg.rref (A [, v])`

Returns the reduced row echelon form of any `m` x `n` matrix `A`.

If a vector `v` is given, the function computes the reduced row echelon form of the augmented matrix `A|v`. In this case, `A` and `v` must have equal dimensions.

See also: **linalg.gsolve**.

```
linalg.scalarmul (v, n)
linalg.scalarmul (n, v)
```

Performs a scalar multiplication by multiplying each element in vector `v` by the number `n`. The result is a new vector.

```
linalg.scale (A)
```

Normalises the (non-null) columns of a matrix `A` in such a way that, in each column, an element of maximum absolute value equals 1. The return is a new matrix where the normalised vectors are delivered in the corresponding columns.

See also: **math.norm**, **stats.scale**.

```
linalg.stack (···)
```

Joins two or more matrices or vectors together vertically. Vectors are supposed to be row vectors. The matrices and vectors must have the same number of columns.

The return is a new matrix.

See also: **linalg.augment**.

```
linalg.submatrix (A, p [, r])
linalg.submatrix (A, p:q [, r:s])
```

In the first form, returns column `p` from matrix `A` as a new row vector.

In the second form, returns columns `p` to `q` as a new matrix.

An optional third argument may be given to limit the extraction of the columns to the specified row `r` or rows `r` to `s`.

With the second and third arguments, you may mix numbers with pairs.

See also: **linalg.column**.

```
linalg.swapcol (A, p, q)
```

Swaps column `p` in matrix `A` with column `q`. `p`, `q` must be positive integers. The result is a new matrix.

See also: **linalg.swaprow**, **linalg.mulrow**, **linalg.mulrowadd**.

```
linalg.swaprow (A, p, q)
```

Swaps row `p` in matrix `A` with row `q`. `p`, `q` must be positive integers. The result is a new matrix.

See also: **linalg.swapcol**, **linalg.mulrow**, **linalg.mulrowadd**.

`linalg.sub (v, w)`

Subtracts vector `w` from vector `v`. The result is a new vector.

See also: **linalg.add**.

`linalg.trace (A)`

Computes the trace of a square matrix `A` and returns a number.

`linalg.transpose (A)`

Computes the transpose of a m x n-matrix `A` and thus returns an n x m-matrix.

`linalg.vector (a1, a2, ···)`

`linalg.vector ([a1, a2, ···])`

`linalg.vector (seq(a1, a2, ···))`

`linalg.vector (n, [a1, a2, ···])`

`linalg.vector (n, [ ])`

Creates a vector with numeric components `a1`, `a2`, etc. The function also accepts a table or sequence of elements `a1`, `a2`, etc. (second and third form).

In the fourth form, `n` denotes the dimension of the vector, and a_k might be single values or key~value pairs. By a metamethod, vector components not explicitly set automatically default to 0. This allows you to create memory-efficient sparse vectors and thus matrices.

In the fifth form, a sparse zero vector of dimension `n` will be returned.

The result is a table of the user-defined type 'vector' and the **linalg.vmt** metatable assigned to allow basic vector operations with the operators **+**, **-**, *****, unary minus and **abs**. The table key `'dim'` contains the dimension of the vector created.

See also: **linalg.matrix**.

`linalg.vaeq (a, b)`

This function checks vector `a` and vector `b` for approximate equality. The return is either **true** or **false**. The function uses Donald Knuth's approximation method to compare vector elements (see the **approx** function for information on how this works).

You can change the accuracy threshold epsilon with the **environ.kernel/eps** function.

See also: ~= metamethod, **approx**, **linalg.veeq**, **linalg.maeq**.

`linalg.veeq (a, b)`

This function checks vector `a` and vector `b`. for strict equality. The return is either **true** or **false**.

See also: == metamethod, **linalg.meeq**, **linalg.vaeq**.

`linalg.vmap (f, v [, ···])`

This operator maps a function `f` to all the components in vector `v` and returns a new vector. The function `f` must return only one value.

If function `f` has only one argument, then only the function and the vector are passed to **linalg.vmap**. If the function has more than one argument, then all arguments *except the first* are passed right after the name of the vector.

Examples:

```
> vmap(<< x -> x^2 >>, vector(1, 2, 3) ):
[ 1, 4, 9 ]

> vmap(<< (x, y) -> x > y >>, vector(1, 0, 1), 0):  # 0 for y
[ true, false, true ]
```

See also: **linalg.vzip**, **linalg.mmap**, **linalg.mzip**.

`linalg.vzip (f, v1, v2 [, ···])`

This function zips together two vectors by applying the function `f` to each of its respective components. The result is a new vector v' where each element v'[k] is determined by v'[k] := f(`v1`[k], `v2`[k]).

`v1` and `v2` must have the same dimension. The third to last argument to `f` must be given right after `v2`.

See also: **linalg.vmap**, **linalg.vzip**, **linalg.mmap**.

`linalg.zerovector (n)`

Creates a zero vector of length `n` with all its components physically set to 0. If you want to create a sparse zero vector of dimension `n`, use: `linalg.vector(n, [])`.

11.14 stats - Statistics

This package contains procedures for statistical calculations and operates completely on tables. As a *plus* package, it is not part of the standard distribution and must be activated with the **import** statement, e.g. `import stats`.

You might want to use **utils.readcsv** to read distributions from a file.

Summary of functions:

Averages:

> **stats.accu, stats.amean, stats.ema, stats.gema, stats.gmean, stats.gsma, stats.gsmm, stats.hmean, stats.iqmean, stats.median, stats.mean, stats.midrange, stats.qmean, stats.sma, stats.smm, stats.trimean, stats.trimmean**.

Combinations:

> **stats.cartprod, stats.numbcomb, stats.numpart, stats.numbperm**.

Deviations:

> **stats.ad, stats.chauvenet, stats.durbinwatson, stats.ios, stats.mad, stats.md, stats.sd, stats.spread, stats.ssd, stats.var**.

Density:

> **stats.cdf, stats.nde, stats.ndf, stats.pdf**.

Extrema:

> **stats.colnorm, stats.extrema, stats.minmax, stats.peaks, stats.rownorm, stats.smallest**.

Occurrences:

> **stats.countentries, stats.freqd, stats.isall, stats.isany, stats.mode, stats.obcount, stats.obpart**.

Ranges:

> **stats.fivenum, stats.iqr, stats.percentile, stats.prange, stats.qcd, stats.quartiles**.

Sums:

> **qsumup, sumup, stats.cumsum, stats.fsum, stats.moment, stats.sumdata, stats.sumdataln, stats.var**.

Probability density functions :

> **stats.cauchy, stats.cdfnormald, stats.chisquare, stats.fratio, stats.gammad, stats.gammadc, stats.invnormald, stats.normald, stats.probit, stats.studentst**.

Miscellaneous:

> **stats.acf, stats.acv, stats.checkcoordinate, stats.dbscan, stats.deltalist, stats.fprod, stats.herfindahl, stats.issorted, stats.kurtosis, stats.neighbours, stats.scale, stats.skewness, stats.sorted, stats.tovals**.

The functions:

A general note: almost all of the statistics functions ignore the **undefined** value should it be part of a distribution. Any non-numeric values in a distribution are replaced with zeros. Most of the following functions also process numarrays.

To reduce round-off errors, it is always a good idea to sort structures before applying **stats** functions that sum up data, see **stats.sorted** and **numarray.sorted**.

```
stats.accu ([true])
```
Returns a factory that computes the running mean, variance, median, and absolute deviation by mere accumulation of individual observations.

If the first optional argument **true** is passed, then sample values are computed (division by the number of observations - 1), otherwise population values are computed (division by the number of observations).

If the resulting factory is called without any argument, the current results are returned in a table.

If the resulting factory is called with a number, i.e. an observation, it is added to the accumulators.

While the mean and variance computed are correct, the median and the absolute deviation are approximations only. The function ignores non-finite values, i.e. **undefined** and +/-**infinity**.

You may use this function if a distribution is too large to be stored in a structure.

The idea has been taken from the Stat package shipped with Digital Equipment Corporation Critical Mass Modula-3. 2.10.1.

`stats.acf (obj, lag, [, option [, m [, s]]])`

Returns the autocorrelation of a distribution `obj` (a table, sequence or numarray) of numbers at a given `lag`, a non-negative integer. If any third argument `option` different from **null** is passed, then the un-normalised autocorrelation will be returned. The return is a number,

$$\sum_{i=1}^{n-lag} (obj_i - \mu)(obj_{i+lag} - \mu)$$

where n is the number of observations, and μ is the arithmetic mean of the distribution. If no `option` is passed, the sum is divided by the variance of `obj` multiplied by n, yielding a normalised result. The function uses Kahan-Ozawa round-off error prevention.

To speed up computation times significantly, you may also pass a precomputed mean `m` and the sum `s` of all values in the distribution.

It may be used to detect periodicy in a time series.

A distribution is autocorrelated if **stats.acf** returns a negative or positive value significantly different from zero. The - normalised - return is in the range [-1, 1], where +1 denotes perfect autocorrelation and -1 with 1 perfect anti-correlation. A negative correlation indicates that higher values of a distribution are related to lower values.

See also: **stats.acv**.

`stats.acv (obj, p, [, option])`

Depending on the type of the observation `obj`, returns a table, sequence or numarray of autocorrelations starting with lag = 0, through and including the given number `p` of lags. If any third argument `option` is passed, then un-normalised autocorrelations are returned. For the formula and numeric method used, see **stats.acf**.

`stats.ad (obj [, option])`

Computes the absolute (or mean) deviation of all the values in a table, sequence or numarray `obj`, i.e. the mean of the equally likely absolute deviations from the arithmetic mean μ:

$$\frac{1}{n} \sum_{i=1}^{n} \left| obj_i - \mu \right|$$

The return is a number.

If any second non-**null** argument is given, then the variation coefficient will be returned:

$$\frac{1}{n} \sum_{i=1}^{n} \left| obj_i - \mu \right| / \left| \mu \right|$$

Absolute deviation is more robust than standard deviation since it is less sensitive to outliers. The function uses Kahan-Babuška round-off error prevention.

If `obj` is empty or entirely consists of **undefined**s, **fail** will be returned. The function ignores **undefined**s, if `obj` features at least one number.

Please note that if `obj` includes non-numbers, where **undefined** is considered a number, they are interpreted as zeros which might unexpectedly influence the result.

The function returns **fail** if `obj` contains less than two elements.

See also: **stats.ios**, **stats.mad**, **stats.md**, **stats.sd**.

`stats.amean (obj)`

Divides each element in a table, sequence or numarray `obj` by the size of `obj` and sums up the quotients to finally return the arithmetic mean. It is equivalent to:

$$\sum_{i=1}^{n} \frac{obj_i}{n}$$

By dividing each element before summation, the function avoids arithmetic overflows and also uses the Kahan-Babuška algorithm to prevent round-off errors during summation. Thus the function is more robust but also significantly slower than **stats.mean**.

If `obj` is table, it is assumed to be an array, non-positive integral keys (including strings, etc.) are ignored.

The function returns **fail** if `obj` contains less than two elements.

If `obj` is empty or entirely consists of **undefined**s, **fail** will be returned. The function ignores **undefined**s, if `obj` features at least one number.

Please note that if `obj` includes non-numbers, where **undefined** is considered a number, they are interpreted as zeros which might unexpectedly influence the result.

See also: **stats.accu**, **stats.gmean**, **stats.hmean**, **stats.mean**, **stats.qmean**, **stats.sma**, **stats.trimmean**.

`stats.bell (n)`

Returns the `n`-th Bell number, i.e. counts the possible partitions of a set.

`stats.cartprod (l)`

Creates a factory that each time it is called, returns a tuple in the Cartesian product of a table of two or more tables `l`, or a sequence of two or more sequences `l`. The type of the result is the same as the type of the input. The structures in `l` may be of different size but must be non-empty.

`stats.cauchy (x, a, b)`

The cauchy[`a`, `b`] distribution has the probability density function:

$$1/(\pi{*}b{*}(1+(\,(x\text{-}a)/b\,)^2)\,), b > 0.$$

See also: **stats.chisquare**, **stats.fratio**, **stats.normald**, **stats.studentst**.

`stats.cdf (a, b [, $\mu$ [, $\sigma$]])`

Computes the cumulative density function between the lower bound `a` and the upper bound `b`. If the mean μ is not given, it defaults to 0; if the standard deviation σ is not given, it defaults to 1.

The return is the number:

$$\frac{1}{\sigma\sqrt{2\pi}} \int_a^b e^{\frac{-(x-\mu)^2}{2\sigma^2}}\, dx$$

See also: **stats.nde**, **stats.ndf**, **stats.pdf**.

`stats.cdfnormald (x)`

Implements the cumulative density function for the standard normal distribution. The return is the number:

$$\frac{1}{\sqrt{2\pi}} \int_{t=-\infty}^{x} e^{\frac{-t^2}{2}} dt$$

See also: **stats.invnormald**, **stats.normald**.

`stats.chauvenet (obj [, x] [, option, ···])`

Receives a table or sequence `obj` of *normally distributed* numbers and checks them for outliers using the formula:

$$p := n * \text{erfc}((| x - \mu | / sd),$$

where n is the number of observations in a distribution, x a sample of it, μ the arithmetic mean $\mu = \sum_{i=1}^{n} \frac{obj_i}{n}$, dev the standard deviation $sd = \sqrt{\frac{1}{n} \sum_{i=1}^{n} \left(obj_i - \mu\right)^2}$.

If at least `obj` and `x` is given, the function checks whether the number `x` is an outlier by conducting a 1-pass check and returns true or false.

If `obj` but not `x` is passed, however, the procedure iterates `obj` again and again as long as it does not find an outlier, and returns the outliers in a structure, its type defined by the type of `obj`.

By default, if p < 0.5, where 0.5 is the magical Chauvenet number, an outlier is detected. If you pass the option `bailout`=c, then c, a non-negative number, will be the threshold.

If you pass the option `jump`=true, as soon as an outlier is detected, it is removed from the distribution and then the whole evaluation process is restarted immediately with a reduced distribution along with a re-computed mean and deviation.

If you do not, all remaining items are also checked according to the current criteria - after the last item has been checked, only then the outliers are removed from the distribution, the mean and deviation are re-computed and another iteration begins.

If you pass the option `mean`=f, where f is a procedure, then the mean μ is determined by f. The default is f = **stats.amean**, i.e. the arithmetic mean.

If you pass the option `dev`=f, where f is a procedure, then the deviation `dev` is determined by f. The default is f = **stats.sd**, the standard deviation.

if you pass the option `outlier='lower'` or `outlier='upper'`, then the function only checks for lower or upper outliers, respectively.

Further information: `Cleaning Data the Chauvenet Way`, by Lily Lin and Paul D. Sherman, published at the South East SAS Users Group's website http://www.sesug.org.

The function is implemented in Agena and included in the lib/stats.agn file.

stats.checkcoordinate (c [, procname])

The function checks whether the given co-ordinate `c` is a pair x:y with both its left-hand and right-hand side x and y being numbers. If a second argument, a string, is given, then error messages of **stats.checkcoordinate** refer to the given procedure `procname` as the function issuing the error. Otherwise the error message includes a reference to **stats.checkcoordinate**.

The function returns the numbers x and y and issues an error otherwise.

stats.chisquare (x, nu)

The chisquare[nu] distribution has the probability density function:

$$x \char`^ ((nu\text{-}2)/2) \exp(\text{-}x/2)/2 \char`^ (nu/2) / \Gamma(nu/2),$$

with $x > 0$ and `nu` a positive integer.

See also: **stats.cauchy**, **stats.fratio**, **stats.normald**, **stats.studentst**.

stats.colnorm (obj)

Returns the largest absolute value of the numbers in the table, sequence or numarray `obj`, and the original value with the largest absolute magnitude. If `obj` includes **undefined**s, they are ignored. If the structure `obj` consists entirely of one or more **undefined**s, then the function returns the value **undefined** twice. If the structure is empty, **fail** will be returned.

See also: **stats.scale**, **stats.rownorm**.

stats.countentries (obj [, f [, ···]])

Counts the number of occurrences of each entry in a table or sequence `obj` and returns a dictionary with its respective key the entry and its value the number of occurrences.

You might optionally pass a procedure `f` to be mapped on the structure before counting begins on the thus modified structure. If `f` has more than one argument, then its *second* to last argument must be given right after `f`.

The function is implemented in Agena and included in the lib/stats.agn file.

See also: **countitems**, **bags** package.

`stats.cumsum (obj)`

Returns a structure of the cumulative sums of the numbers in the table, sequence or numarray `obj`.

The type of return is determined by the type of `obj`.

The function returns **fail** if `obj` contains less than one element. It may also return a structure containing **undefined** and/or **infinity** if `obj` includes non-numbers.

See also: **sumup**, **calc.fsum**, **stats.fsum**, **stats.sumdata**.

`stats.dbscan (obj, eps, minpts [, option])`

The functions finds clusters in a sequence `obj` of n-dimensional points and returns a table with the individual clusters along with their respective points.

It also returns a register of the size of the whole distribution listing the cluster number associated with each point, where the point in this case is represented by its integral position in the sequence `obj`.

The co-ordinates of points in `obj` may be represented by pairs (2-dimensional space, only), sequences (any space), or vectors created by **linalg.vector** (any space).

`eps` is the maximum allowed distance between two points that shall belong to the same neighbourhood. `minpts` is the minimum number of points that shall constitute a neighbourhood.

By specifying the `'select'` option along with a function returning a Boolean, e.g. `'select':<< x -> right x < 1 >>`, only points satisfying the given criterion are examined.

By specifying the `'method'` option, you can control how the function determines clusters: `'method':'original'` uses the classic one, `'method':'modified'` uses a much faster and memory-saving implementation that contrary to the original method immediately flags neighbours of neighbours as being visited and thus does not examine them again in further passes. The default is `'original'`.

See also: **stats.neighbours**.

`stats.deltalist (obj [, option])`

Returns a structure of the deltas of neighbouring elements in the table, sequence or numarray `obj`. If the value **true** is given as an option, then absolute differences are returned.

The type of return is determined by the type of `obj`.

Please note that the difference between **undefined** and a number is **undefined**, and that the difference between **infinity** and a number is ±**infinity**.

The function returns **fail** if `obj` contains less than two elements.

See also: **stats.ios**.

`stats.durbinwatson (obj)`

The Durbin-Watson test detects the autocorrelation in the residuals from a linear regression and returns

$$d = \sum_{i=2}^{n}\left(obj_i - obj_{i-1}\right)^2 / \sum_{i=1}^{n} obj_i^2$$

If d is equal to 2, it indicated the absence of autocorrelation. If d is less than 2, it indicates positive autocorrelation; if d is greater than 2 it indicates negative autocorrelation and that the observations are very different from each other. If d is less than 1, the regression should be checked. The function uses Kahan-Babuška roundoff prevention. `obj` may be a table, sequence or numarray.

`stats.ema (obj, k, alpha [, mode [, y0star]])`

Computes the exponential moving average of a table or sequence `obj` up to and including its `k`-th element.

The smoothing factor `alpha` is a rational number in the range [0, 1].

The function supports two algorithms: If `mode` is 1 (the default), then the algorithm

```
r := alpha * obj[k];
s := 1 - alpha;
for i from k - 1 to 1 by -1 do
   r +:= alpha * s ^ i * obj[i]
od;
r := r + s ^ k * y0star;
```

is used to compute the result r. In `mode` 1, you can pass an explicit first estimate `y0star`, otherwise the first value `y0star` is equal to the sample moving average of `obj`. If `mode` is 2, then the formula

```
r := obj[k];
for i from k - 1 to 1 by -1 do
   r +:= alpha * (obj[i] - r)
od;
```

is applied.

The result is a number.

See also: **stats.gema**.

`stats.extrema (obj, delta)`

Expects a sequence or table `obj` of points $x_k{:}y_k$ and the number `delta` and determines the local minima and maxima.

A value y_k is considered an extrema if the difference to its surrounding is at least `delta`. The function returns two structures of pairs, i.e. points, the first one including the local minima, the second one the local maxima.

The type of the structures is determined by the type of `obj`.

The function is implemented in Agena and included in the lib/stats.agn file.

`stats.fivenum (obj)`

Returns a sequence of the first quartile, the median, and the third quartile of a distribution `obj`, in this order. `obj` may be a table, sequence or numarray. If the number of observations is five or more, the sequence also includes the minimum and the maximum observation, along with the arithmetic mean.

The first and third quartiles are computed according to the NIST rule, see **stats.percentile** for further information.

If the elements in `obj` are not sorted in ascending order, the function automatically sorts them non-destructively, and any non-numeric values are converted to zeros.

See also: **stats.quartiles**.

`stats.fprod (f, obj [, a [, b [, ···]]])`

Applies the function `f` onto all elements in the table, sequence or numarray `obj` and then multiplies the results. The return is the number:

$$\prod_{i=a}^{b} f(obj_i)$$

If `a` is not given, `a` is set to 1. If `b` is not given, `b` is set to the number of elements in `obj`. If `f` is a multivariate function, its second, third, etc. argument must be passed after `b`.

See also: **calc.fsum**, **stats.fsum**, **stats.sumdata**.

stats.fratio (x, nu1, nu2)

The Fisher's F distribution, also known as fratio distribution, has the probability density function

$$\Gamma(\ (\texttt{nu1}+\texttt{nu2})/2\ )\ /\ \Gamma(\texttt{nu1}/2)/\Gamma(\texttt{nu2}/2)*(\texttt{nu1}/\texttt{nu2})\char`^(\texttt{nu1}/2)\ *$$
$$\texttt{x}\char`^((\texttt{nu1}-2)/2)\ /\ (1+(\texttt{nu1}/\texttt{nu2})*\texttt{x})\char`^((\texttt{nu1}+\texttt{nu2})/2)$$

with $x > 0$, `nu1` and `nu2` positive integers.

See also: **stats.cauchy**, **stats.chisquare**, **stats.normald**, **stats.studentst**.

stats.freqd (s, p [, n])
stats.freqd (s, p [, option])

For sample `s`, the function computes a frequency distribution function that each time it is called, returns both the start of the respective subinterval (not the class number) defined by pair `p` and step size or number of classes and the number of occurrences in this subinterval/class. If the distribution has been completely traversed, two **nulls** are returned.

For more information on the arguments to be passed and the values returned, please refer to the description of **stats.obcount**.

Example:

```
> s := seq(-1, 0, 0.1, 0.2, 0.3, 0.4, 1, 1.1, 2, 2.1)

> stats.obcount(s, 0:2, 0.5):
[0 ~ 5, 0.5 ~ 0, 1 ~ 2, 1.5 ~ 1]          [-1, 2.1]

> f := stats.freqd(s, 0:2, 0.5);

> f():
0        5

> f():
0.5      0
```

The function is implemented in Agena and included in the lib/stats.agn file.

See also: **stats.obcount**.

`stats.fsum (f, obj [, a [, b [, ···]]])`

Applies the function `f` onto all elements in the table, sequence or numarray `obj` and then sums up the results using Kahan-Babuška round-off error prevention. The return is the number:

$$\sum_{i=a}^{b} f(obj_i)$$

If `a` is not given, `a` is set to 1. If `b` is not given, `b` is set to the number of elements in `obj`. If `f` is a multivariate function, its second, third, etc. argument must be passed after `b`.

See also: **calc.fsum**, **stats.fprod**, **stats.sumdata**.

`stats.gammad (x, a, b)`

The Gamma distribution function returns the integral from zero to real `x` of the gamma probability density function and returns the number:

$$\frac{a^b}{\Gamma(b)} \int_{0}^{x} t^{b-1} \, e^{-at} \, dt$$

where `a` * `x` > 0, `b` > 0. See also: **stats.gammadc**.

`stats.gammadc (x, a, b)`

The complemented Gamma distribution function returns the integral from x to infinity of the gamma probability density function and returns the number:

$$\frac{a^b}{\Gamma(b)} \int_{X}^{\infty} t^{b-1} \, e^{-at} \, dt$$

where `a` * `x` > 0, `b` > 0. See also: **stats.gammadc**.

`stats.gema (obj, k, alpha [, mode [, y0star]])`

Like **stats.ema**, but returns a function that, each time it is called, returns the exponential moving average, starting with sample `obj`[k], and progressing with sample `obj`[k+1], `obj`[k+2], etc. with subsequent calls. It return **null** if there are no more samples in `obj`. It is much faster than **stats.ema** with large distributions.

The smoothing factor `alpha` is a rational number in the range [0, 1].

The function supports two algorithms: If `mode` is 1 (the default), then the algorithm

```
r := alpha * obj[k];
s := 1 - alpha;
for i from k - 1 to 1 by -1 do
    r := r + alpha * s ^ i * obj[i]
od;
r := r + s ^ k * y0star;
```

is used to compute the result. In `mode` 1, you can pass an explicit first estimate `y0star`, otherwise the first value `y0star` is equal to the sample moving average of `obj`.

If `mode` is 2, then the formula

```
r := obj[k];
for i from k - 1 to 1 by -1 do
    r := r + alpha * (obj[i] - r)
od;
```

is applied to the period.

The result is a number.

stats.gini (obj [, 'sorted'])

Measures the inequality in a distribution given by the table, sequence or numarray `obj` by applying Gini's formula

$$\sum_{i=1}^{n} \sum_{j=1}^{n} |x_i - x_j| \ / \ 2n^2\mu,$$

where n is the number of occurrences and μ the arithmetic mean.

All members of `obj` should be numbers. **infinity**'s or **undefined**'s are ignored.

It returns a number r indicating the absolute mean of the difference between every pair of observations, divided by the arithmetic mean of the population, with $0 \leq r \leq 1$, where 0 indicates that all observations are equal, and (a theoretical value of) 1 indicates complete inequality. It is assumed that all observations are non-negative.

If the option `'sorted'` is given then the function assumes that all elements in `obj` are already sorted in ascending order - thus computing the result much faster.

To compute the normalised Gini coefficient, multiply the result by n/(n-1).

See also: **stats.herfindahl**.

`stats.gmean (obj)`

Returns the geometric mean of all numeric values in table, sequence or numarray `obj`. It is a measure of central tendency. Its formula is:

$$\left(\prod_{i=1}^{n} obj_i \right)^{1/n}$$

The function returns **fail** if `obj` contains less than two elements.

The geometric mean should be applied on positive values that are interpreted to their products, e.g. rates of growth, instead of their sums, only. Otherwise, **undefined** may be returned.

The function is implemented in Agena and included in the lib/stats.agn file.

See also: **stats.amean**, **stats.hmean**, **stats.mean**, **stats.qmean**.

`stats.gsma (obj, k, p)`
`stats.gsma (obj, k, p, b)`

Like **stats.sma**, but returns a function that, each time it is called, returns the simple moving mean, starting with sample `k`, and progressing with sample `k+1`, `k+2`, etc. If `k` > **size** `obj`, then the function returns **null**. It is much faster than **stats.sma** with large distributions.

`stats.gsmm (obj, k, p)`
`stats.gsmm (obj, k, p, b)`

Like **stats.smm**, but returns a function that, each time it is called, returns the simple moving median, starting with sample `k`, and progressing with sample `k+1`, `k+2`, etc. If `k` > size(`obj`), then the function returns **null**. It is much faster than **stats.smm** with large distributions.

The function automatically non-destructively sorts the distribution `obj` if it is unsorted.

`stats.herfindahl (obj)`

Returns the normalised Herfindahl–Hirschman index of a distribution `obj` (a table, sequence or numarray), an indicator of the amount of competition in economy. A value of 0 means that there is absolute competition, i.e. that all companies have the same share, and 1 means that there is a monopoly.
The normalised index h is defined as:

$$H = \sum_{i=1}^{n} \left(\frac{obj_i}{s} \right)^2 \text{, where } s = \sum_{i=1}^{n} obj_i, \Rightarrow h = \frac{H - 1/n}{1 - 1/n}$$

It is also a good measure to determine the stability of a distribution, with a value tending to zero indicating that the number of outliers is quite low, and a value tending to 1 that there is at least an extreme outlier.

The function is implemented in Agena and included in the lib/stats.agn file.

See also: **stats.gini**.

`stats.hmean (obj)`

Returns the harmonic mean of all numeric values in table, sequence or numarray `obj` as a number. It is useful with rates and ratios, as it provides the best average. It is defined as follows:

$$n \Big/ \sum_{i=1}^{n} \frac{1}{obj_i}$$

The function returns **fail** if `obj` contains less than two elements.

The harmonic mean should be applied on observations containing relations to a unit, e.g. speed.

The function internally uses Kahan-Babuška summation to compensate for rounding errors.

See also: **stats.amean**, **stats.gmean**, **stats.mean**, **stats.qmean**.

`stats.invnormald (y)`

Evaluates the inverse of the Normal distribution function by returning the argument, x, for which the area under the Gaussian probability density function (integrated from $-\infty$ to x) is equal to `y`.

See also: **stats.cauchy**, **stats.chisquare**, **stats.fratio**, **stats.normald**, **stats.studentst**.

`stats.ios (obj [, option])`

Sums up absolute differences between neighbouring entries in a table, sequence or numarry `obj`, divides by the number of its elements minus 1, and returns the number:

$$\frac{1}{n-1} \sum_{i=2}^{n} \left| obj_i - obj_{i-1} \right|$$

The function returns **fail** if `obj` contains less than two elements.

If any second non-**null** argument is given, the function first normalises the distribution to the range (-∞, 1] (see **stats.scale**), determines the difference list, sums up its absolute differences and divides the sum by the number of occurrences minus 1 to make a distribution comparable to other ones.

This indicator is quite useful to find out how stable or volatile a preferably unsorted distribution is.

See also: **stats.ad**, **stats.deltalist**, **stats.sd**, **stats.var**.

`stats.iqmean (obj)`

Returns the arithmetic mean of the interquartile range of the distribution `obj` using Kahan-Babuška round-off error prevention. The return is a number. `obj` may be a table, sequence or numarray.

If a distribution is unsorted, the function automatically sorts it non-destructively, and any non-numeric observations are converted to zeros.

The interquartile range comprises all observations that reside between the first and third quartiles.
See also: **stats.iqr**, **stats.midrange**.

`stats.iqr (obj [, a [, b]])`

Without `a` and `b` given, the function determines the interquartile range (IQR), i.e. the difference of the third and first quartile. **stats.iqr** is useful for determining the variability in a distribution `obj` (a table, sequence or numarray).

You may optionally pass a lower and upper percentile `a`, `b`, both in the range [0, 100). If a is missing, it is set to 25. If `b` is missing it is set to 100 - `a`.

It returns the number

$$\textbf{stats.percentile}(\texttt{obj, b}) - \textbf{stats.percentile}(\texttt{obj, a})$$

If `obj` is unsorted, the function sorts it non-destructively. It is implemented in Agena and included in the lib/stats.agn file.

See also: **stats.midrange**, **stats.percentile**, **stats.qcd**, **stats.quartiles**.

`stats.isall (obj [, eps])`

Checks whether all elements in a table or sequence `obj` are non-zero and returns **true** or **false**. If the second argument `eps`, a non-negative number, is passed, the function returns **true** if all observations x in `obj` satisfies the condition abs(x) > `eps`. By default `eps` is 0.

The function returns **fail** with empty structures or structures that contain one element only.

See also: **and** operator, **stats.isany**.

`stats.isany (obj [, eps])`

Checks whether at least one element in a table or sequence `obj` is non-zero and returns **true** or **false**. If the second argument `eps`, a non-negative number, is passed, the function returns **true** if at least one observations x in `obj` satisfies the condition abs(x) > `eps`. By default `eps` is 0.

The function returns **fail** with empty structures or structures that contain one element only.

See also: **or** operator, **stats.isall**.

`stats.issorted (obj [, f])`

Checks whether all values in a table, sequence or numarray `obj` of numbers are stored in ascending order and returns **true** or **false**. If a value in `obj` is not a number, it is ignored.

If `obj` is a table, you have to make sure that it does not contain holes. If it contains holes, apply **tables.entries** on `obj`.

If `f` is given, then it must be a function that receives two structure elements to determine the sorting order. See **sort** for further information.

See also: **sort**, **sorted**, **skycrane.sorted**, **stats.sorted**.

`stats.kurtosis (obj)`

The function determines the kurtosis, a measure of flatness or peakedness of symmetric and unimodal distributions.

To quote Wikipedia, a higher value means that the distribution has `a sharper peak and fatter tails,` while a lower value indicates `the distribution has a more rounded peak and thinner tails.`

The function computes the result by computing the fourth moment around the mean of a distribution, divided by the fourth power of the standard deviation.

The function returns **fail** if `obj` contains less than two elements.

The function is implemented in Agena and included in the lib/stats.agn file.

See also: **stats.skewness**.

`stats.mad (obj [, option])`

Returns the median of the absolute deviations of all numeric values in table, sequence or numarray `obj` from `obj`'s median, and returns the number:

$$\text{stats.median}(\bigvee_{i=1}^{\text{size obj}} \left| \text{obj}_i - \text{stats.median(obj)} \right|).$$

If any second non-**null** argument is given, then the variation coefficient will be returned:

$$\text{stats.median}(\bigvee_{i=1}^{\text{size obj}} \left| \text{obj}_i - \text{stats.median(obj)} \right|) / \text{stats.median(obj)}.$$

Median absolute deviation is quite robust if a distribution contains a small number of outliers.

If `obj` is unsorted, it automatically sorts it before determining the result.

If `obj` contains less than two elements or entirely consists of **undefined**s, **fail** will be returned. The function ignores **undefined**s, if `obj` features at least one number.

Please note that if `obj` includes non-numbers, where **undefined** is considered a number, they are interpreted as zeros which might unexpectedly influence the result.

See also: **stats.ad**, **stats.md**, **stats.median**.

`stats.md (obj [, option])`

Computes the median deviation of all the values in a table, sequence or numarray `obj`, i.e. the mean of the equally likely absolute deviations from the median med:

$$\frac{1}{n} \sum_{i=1}^{n} \left| \text{obj}_i - \text{med} \right|$$

The return is a number.

If any second non-**null** argument is given, then the variation coefficient will be returned:

$$\sqrt{\frac{1}{n} \sum_{i=1}^{n} \left| \text{obj}_i - \text{med} \right| } \; / \; | \text{ med } |$$

See also: **stats.mad**.

`stats.mean (obj)`

Returns the arithmetic mean of all numeric values in table, sequence or numarray `obj` as a number. It is equivalent to:

$$\frac{1}{n}\sum_{i=1}^{n} obj_i$$

thus the function - as opposed to **stats.amean** - first computes the sum of the observations and then divides it by the number of elements.

If `obj` is table, it is assumed to be an array, non-positive integral keys (including strings, etc.) are ignored.

The function returns **fail** if `obj` contains less than two elements.

For a more robust but slower version, please have a look at **stats.amean**.

The function is implemented in Agena and included in the lib/stats.agn file.

See also: **stats.accu**, **stats.amean**, **stats.gmean**, **stats.hmean**, **stats.meanmed**, **stats.qmean**.

`stats.meanmed (obj [, option])`

Returns both the arithmetic mean and the median of all numeric values in table, sequence or numarray `obj` as numbers. If any `option` is given, the quotient of the mean and the median will be returned.

See also: **stats.accu**, **stats.amean**, **stats.meanvar**, **stats.median**.

`stats.meanvar (obj [, option])`

Returns both the arithmetic mean and the variance - in this order - of the distribution `obj` using an algorithm developed by B. P. Welford to prevent round-off errors.

By default, the population variance will be returned unless you pass the Boolean value **true** for `option` to compute the sample variance. `obj` may be a table, sequence or numarray.

See also: **stats.meanmed**.

`stats.median (obj)`

Returns the median of all numeric values in table, sequence or numarray `obj` as a number. If `obj` is unsorted, it automatically sorts it before determining the median.

If `obj` contains less than two elements or entirely consists of **undefined**s, **fail** will be returned. The function ignores **undefined**s, if `obj` features at least one number.

Please note that if `obj` includes non-numbers, where **undefined** is considered a number, they are interpreted as zeros which might unexpectedly influence the result.

The median is the middle element of a distribution if its size is odd, or the average of its middle elements it is size is even.

See also: **stats.accu**, **stats.mad**, **stats.meanmed**.

`stats.midrange (obj [, option])`

Returns both the arithmetic mean and the variance - in this order - of the distribution `obj`, a table, sequence or numarray.

Computes the sum of the minimum and maximum value of a distribution `obj`, divided by two.

If the option `'sorted'` is given, the observation is not traversed; instead the first and the last entry is taken to compute the mean. If the observation is empty or has only one element, **fail** will be returned.

See also: **stats.iqr**, **stats.minmax**.

`stats.minmax (obj [, 'sorted'])`

Returns a table with the minimum of all numeric values in table, sequence or numarray `obj` as the first value, and the maximum as the second value. If the option `'sorted'` is passed than the function assumes that all values in `obj` are sorted in ascending order so that execution is much faster.

stats.minmax returns **fail** if a sequence or table of less than two elements has been passed. If `obj` consists entirely of **undefined** entries, $[-\infty, \infty]$ or **seq**$(-\infty, \infty)$ are returned.

See also: **stats.midrange**.

`stats.mode (obj)`

Returns all values in the sequence or table `obj` with the largest number of occurrence, i.e. highest frequency. If there is more than one value with the highest frequency, they are all returned.

The type of return is determined by the type of its argument. If the given structure is empty, it is simply returned.

The function is implemented in Agena and included in the lib/stats.agn file.

`stats.moment (obj [, p [, xₘ [, option]]])`

Computes the moment `p` of the given table, sequence or numarray `obj` about any origin `xₘ` for a full population and returns a number. It is equivalent to:

$$\frac{1}{n} \sum_{i=1}^{n} \left(obj_i - x_m\right)^p$$

If only `obj` is given, the moment `p` defaults to 1, and the origin `xₘ` defaults to 0. If given, the moment `p` and the origin `xₘ` must be numbers. If `obj` contains less then two observations, **fail** will be returned.

if `option` is given and is **true**, the sample moment

$$\frac{1}{n-1} \sum_{i=1}^{n} \left(obj_i - x_m\right)^p$$

is computed.

See also: **qmdev**, **stats.sumdata**.

`stats.nde (x [, [μ, [σ]])`

Computes $e^{\frac{-(x-\mu)^2}{2\sigma^2}}$; μ and σ default to 0 and 1, respectively.

See also: **stats.nd**f, **stats.pdf**.

`stats.ndf ([σ])`

Computes $\dfrac{1}{\sqrt{2\pi}}$ if σ is not given, and $\dfrac{1}{\sigma\sqrt{2\pi}}$ otherwise, and issues an error if $\sigma \leq 0$.

See also: **stats.nde**, **stats.pdf**.

`stats.neighbours (obj, idx, eps [, power [, indices]])`

Determines all neighbours of a given n-dimensional point in a distribution `obj` that lie in a certain Euclidian distance `eps`. `idx` is the position of the point of interest in the distribution - a positive integer -, and not the point itself. `eps` is any positive number, `power` is a positive integer with which the respective Euclidean distances and `eps` shall be raised before a comparison is conducted, its default is 2.

The return is a sequence with the nearby points. If the fifth argument `indices` is **true**, however, then not the points but their positions in the distribution are returned.

The points may be represented either as pairs (2-dimensional space), sequences of co-ordinates (n-dimensional space), or any n-dimensional vectors created by the **linalg.vector** function.

See also: **linalg.norm**, **stats.dbscan**.

`stats.normald (x [, ` μ ` [, ` σ `]])`

The normal distribution has the probability density function:

$$\exp(\ -(x - \mu)^2 \ / \ 2 \ / \ \sigma^2\ ) \ / \ \sqrt{2\pi\sigma^2}$$

σ is the standard deviation and must be positive. μ defaults to 0, and σ to 1.

See also: **stats.cauchy**, **stats.chisquare**, **stats.fratio**, **stats.invnormald**, **stats.studentst**.

`stats.numbcomb (n, r)`
`stats.numbcomb (s, r)`

In the first form, counts the number of combinations of `n` things taken `r` at a time. In the second form, the function counts the number of combinations all the elements in the set `s` taken `r` at a time. The set may include data of any type.

If `n` or `r` are non-integral or negative, the function returns **undefined**.

The function is implemented in Agena and included in the lib/stats.agn file.

See also: **binomial**, **fact**, **stats.numbperm**.

`stats.numbpart (n, r)`

Computes the number of partitions of `n`, the partition numbers, taken `r` at a time. By default, `r = n`.

```
stats.numbperm (n, r)
stats.numbperm (s, r)
```

In the first form, counts the number of permutations of `n` things taken `r` at a time. In the second form, the function counts the number of permutations of all the elements in the set `s` taken `r` at a time. The set may include data of any type.

If `n` or `r` are non-integral or negative, the function returns **undefined**.

The function is implemented in Agena and included in the lib/stats.agn file.

See also: **binomial**, **fact**, **stats.numbcomb**.

```
stats.obcount (s, p, n [, f])
stats.obcount (s, p [, option])
```

The function counts occurrences in an observation.

In the first form, it first divides a numeric range defined by the pair `p` and its step size `n` into its respective classes.

In the second form, if the option is `classes=`k, it first divides a numeric range defined by the pair `p` into k classes. If no option is passed, it automatically computes the number of classes according to the formula

$$1 + 3.3*\ln(\text{size of range } p),$$

with no upper limit. (It is suggested to choose between 5 or 30 classes.)

With both forms, all occurrences in the distribution `s` (a sequence) are then sorted into these subranges/classes and the function finally counts all elements in them. If the optional fourth argument `f`, a function, is given, then an occurrence or a part of an occurrence is first converted according to the function definition before the correct subinterval is being determined.

The function returns a table with the keys the respective left borders of the subranges and the values the number of counts in the respective subranges. It always also returns a second table which may include all those elements in `s` which are not part of the overall range defined by `p`. If all numbers in `s` fit into `p`, an empty table will be returned.

If an element in `s` equals the right border of a subinterval, then it is considered to be part of the next subinterval. But if an element in `s` equals the right border of the overall interval `p`, it is considered part of the last subinterval.

The function issues an error if it encounters a non-number in `s`, or if the left border in `p` is greater or equals to the right border in `p`.

The function is implemented in Agena and included in the lib/stats.agn file.

An example:

```
> s := seq(0.1, 0.2, 0.3, 0.4, 1, 1.1, 2, 2.1);

> stats.obcount(s, 0:2, 1):
[0 ~ 4, 1 ~ 3]   [2.1]
```

See also: **stats.freqd**, **stats.obpart**.

stats.obpart (s, p, n [, f [, g]])
stats.obpart (s, p, option [, f [, g]])
The function sorts occurrences into subintervals (classes).

In the first form, it divides a numeric range defined by the pair `p` and its step size `n` into its respective subintervals, and sorts all occurrences in the distribution `s` (a sequence) into these classes.

In the second form, if the option is `class=`k, divides a numeric range defined by the pair `p` into k classes. If the option **null**, the function automatically computes the number of classes according to the formula

$$1 + 3.3*\ln(\text{size of range } p),$$

with no upper limit. (It is suggested to choose between 5 or 30 classes.)

If the fourth argument `f`, a function, is given, then an occurrence or a part of an occurrence is first converted according to the function definition before the correct subinterval is being determined. If `f` is **null**, no conversion is done.

If the fifth argument `g`, a function, is given, then it is applied on an occurrence or part of it before it is inserted into the subinterval that already has been determined.

The function returns a table with the keys the respective left borders of the subranges and the values sequences with the respective occurrences. It always also returns a second table which may include all those elements in `s` which are not part of the overall range defined by `p`.

If an element in `s` equals the right border of a subinterval, then it is considered to be part of the next subinterval. But if an element in `s` equals the right border of the overall interval `p`, it is considered part of the last subinterval.
The function issues an error if a distribution or part of it is not or could not be converted to a number, or if the left border in `p` is greater or equals to the right border in `p`.

The function is implemented in Agena and included in the lib/stats.agn file.

See also: **stats.obcount**.

Examples:

```
> s := seq(1.1, 1.2, 2.4, 2.5, 2.6, 3.1);

> stats.obpart(s, 1:4, 1):
[seq(1.1, 1.2), seq(2.4, 2.5, 2.6), seq(3.1)]    []
```

Given are timestamps and running times in seconds:

```
> s := seq('12:30:05.017':3, '12:31:57.235':4);
```

To convert a timestamp into its decimal representation, so that **stats.obpart** can sort an occurrence into a subinterval, we define the following function:

```
> import clock

> f := proc(x) is
>    local hrs, min, sec;
>    hrs, min, sec :=
>       strings.match(left(x), '(%d%d):(%d%d):(%d%d\.%d%d%d)');
>    return clock.todec(clock.tm(  # returns a number
>       tonumber(hrs), tonumber(min), tonumber(sec)))
> end;

> stats.obpart(s, 12.4:12.6, 1/60, f):
[12.4 ~ seq(), ..., 12.5 ~ seq(12:30:05.017:3),
 12.516667 ~ seq(12:31:57.235:4), ...]    []
```

We only want to insert the running times in milliseconds, but not the timestamps:

```
> g := << x -> right(x)*1k >>;

> stats.obpart(s, 12.4:12.6, 1/60, f, g):
[12.4 ~ seq(), ..., 12.5 ~ seq(3000), 12.516667 ~ seq(4000), ...]    []
```

See also: **stats.obcount**.

stats.pdf (x [, μ [, σ]])

Computes the probability density function for the normal distribution at the numeric value x. The defaults are $\mu = 0$, with standard deviation $\sigma = 1$, thus determining the standard normal distribution.

The return is the number:

$$\frac{1}{\sigma\sqrt{2\pi}}\, e^{\frac{-(x-\mu)^2}{2\sigma^2}}$$

See also: **stats.cdf**, **stats.nde**, **stats.ndf**.

`stats.peaks (obj, delta [, dv])`

The function returns all peaks and valleys of a distribution `obj` consisting of two-dimensional numeric co-ordinates represented as pairs xk:yk. `obj` may be a table or sequence. A point is considered an extremum if the `vertical` difference to its surrounding is at least `delta`, a positive number. By default, if `dv` is not given or is 1, the direct neighbours of each point are considered, otherwise the `dv`-th neighbours to the left and the right of each point are checked.

Depending on the type of o, the first return is a structure including all valleys represented as pairs xk:yk, and the second return is a structure of the peaks as pairs xk:yk.

See also: **stats.extrema**.

`stats.percentile (obj, p [, option])`

Returns the value below which a certain percent `p` of the elements in `obj` fall.

`obj` must be a table, sequence or numarray, `p` an integer in the range $0 \leq p < 100$. If no option is given, then the percentile is determined by computing the nearest rank (rank = p/100 * size `obj` + ½, `Wikpedia method`). If `option` is the string `'nist'`, then the method proposed by NIST is used (rank = p/100 * (size `obj` + 1)); if the string `'excel'` is given for `option`, then the algorithm used by Excel is used (rank = p/100*(size `obj` -1) + 1).

The function issues an error if `obj` is empty. It is implemented in Agena and included in the lib/stats.agn file.

See also: **whereis**, **stats.quartiles**.

`stats.prange (obj [, a [, b]])`

Returns all elements in a table, sequence or numarray `obj` from the `a`-th percentile rank up but not including the `b`-th percentile rank. `a` and `b` must be positive integers in the range [0 .. 100). If `a` and `b` are not given, `a` is set to 25, and `b` to 75. If `b` is not given, it is set to 100 - `a`. The type of return is determined by the type of `obj`. If the elements in `obj` are not sorted in ascending order, the function automatically sorts them non-destructively, and any non-numeric values are converted to zeros.

`stats.probit (x)`

Implements the quantile function associated with the standard normal distribution, or in other words: the inverse of the cumulative distribution function of the standard normal distribution for number `x`. It returns the number $\sqrt{2}\ \mathrm{erf}^{-1}(2x–1)$.

`stats.qcd (obj [, a [, b]])`

Without `a` and `b` given, the function determines the interquartile range (IQR) of a distribution `obj` (a table, sequence or numarray), i.e. the difference of the third (= Q3) and first (= Q1) quartile divided by the sum of the third and first quartile:

$$\frac{Q_3 - Q_1}{Q_3 + Q_1}.$$

You may optionally pass a lower and upper percentile `a`, `b`, both in the range [0, 100). If a is missing, it is set to 25. If `b` is missing it is set to 100 - `a`.

If `obj` is unsorted, the function sorts it non-destructively. It is implemented in Agena and included in the lib/stats.agn file.

See also: **stats.iqr**, **stats.percentile**, **stats.quartiles**.

`stats.qmean (obj)`

Returns the quadratic mean (root mean square) of all numeric values in table, sequence or numarray `obj` as a number. If `obj` is table, it is assumed to be an array, non-positive integral keys (including strings, etc.) are ignored. It can be used to measure the magnitude of a quantity which variates are positive and negative, e.g. sinusoids.

It is equivalent to:

$$\sqrt{\frac{1}{n} \sum_{i=1}^{n} obj_i^2}$$

The function returns **fail** if `obj` contains less than two elements.

The function is implemented in Agena and included in the lib/stats.agn file.

See also: **stats.amean**, **stats.gmean**, **stats.hmean**, **stats.mean**.

`stats.quartiles (obj)`

`stats.quartiles (obj [, pos])`

In the first form, it returns the first, second, and third quartile of table, sequence or numarray `obj`. The first and third quartiles are computed according to the NIST rule, see **stats.percentile** for further information.

It also determines the lower outlier limit L_1, where L_1 = first quartile - 1.5 times the interquartile range of `obj`, and the upper outlier limit U_1, where U_1 = third quartile + 1.5 times the interquartile range of `obj`. If a value x in `obj` is equal to L_1 or U_1, then x will be returned. If L_1 is not included in `obj`, then the next largest value to L_1 will be returned. If U_1 is not included in `obj`, then the next smallest value to U_1 is computed.

Finally it computes the interquartile range, i.e. third quartile - first quartile. The order is: first quartile, median, third quartile, `L₁`, `U₁`, and the interquartile range.

In the second form, if either the integer 1, 2, or 3 is passed for the optional second argument `pos`, the first, second, or third quartile will be returned as a number, respectively.

If the elements in `obj` are not sorted in ascending order, the function automatically sorts them non-destructively, and any non-numeric values are converted to zeros.

The number of values in `obj` should be at least 12, better are 20 or more values. if the number of values is less than 2, **fail** will be returned.

See also: **whereis**, **stats.fivenum**, **stats.iqr**, **stats.percentile**, **stats.qcd**.

`stats.rownorm (obj)`

Returns the sum of the absolute values of the numbers in the table, sequence or numarray `obj`. If `obj` includes **undefined**s, they are ignored. If the structure consists *entirely* of one or more **undefined**s, then the function returns **undefined**. If the structure is empty, **fail** will be returned.

See also: **stats.scale**, **stats.colnorm**.

`stats.scale (obj [, option])`

The procedure normalises the numbers in the table or sequence `obj` in such a way that an element of maximum absolute value equals 1, thus scaling a distribution to the range (−∞, 1] by dividing all observations by this maximum element.

When given a second option, the function normalises all its observations to the range [0, 1]. See **math.norm** for further details.

The normalised numbers are returned in a new table or sequence, depending on the type of `obj`.

If the maximum absolute value is 0, the function returns **fail**.

See also: **math.norm**, **linalg.scale**.

`stats.sd (obj [, sample [, option]])`

Returns the standard deviation of all numeric values in table, sequence or numarray `obj` as a number. If `obj` is a table, it is assumed to be an array, non-positive integral keys (including strings, etc.) are ignored.

If `sample` is not given or is not **true**, it returns the population standard deviation:

$$\sigma = \sqrt{\frac{1}{n} \sum_{i=1}^{n} \left(obj_i - \mu\right)^2}$$

where μ is the arithmetic mean of a distribution.

If `sample` is given and is **true**, the (unbiased) sample standard deviation will be returned:

$$\sigma = \sqrt{\frac{1}{n\text{-}1} \sum_{i=1}^{n} \left(obj_i - \mu\right)^2}$$

If the return is a small number, it indicates that the points in a distribution are close to its mean m. A large value indicates that its points are rather spread out. Contrary to variance, standard deviation is expressed in the same units as the data. Standard deviation is less robust to outliers than absolute deviation.

The function returns **fail** if `obj` contains less than two elements.

If any third non-**null** argument is given, then the coefficient $\sigma / |\mu|$ will be returned to make different distributions comparable.

The function is implemented in Agena and included in the lib/stats.agn file.

See also: **qmdev**, **stats.ad**, **stats.chauvenet**, **stats.ios**, **stats.mad**, **stats.var**.

`stats.skewness (obj)`

Returns the sample skewness, a measure of the asymmetry of the probability distribution represented by the table or sequence `obj` of numbers. Returns 0 if a distribution is symmetric, a negative value if the left tail is longer, and a positive value if the right tail is longer.

It computes the third moment about the mean and divides it by the third power of the standard deviation.

The function returns **fail** if `obj` contains less than two elements.

The function is implemented in Agena and included in the lib/stats.agn file.

See also: **stats.kurtosis**.

```
stats.sma (obj, k, p)
stats.sma (obj, k, p, b)
```

In the first form, computes the simple moving average of a table or sequence `obj` by averaging the last `p` numbers from the structure (`p` is also known as the `period`) including sample `k`, i.e.:

$$\frac{1}{p} \sum_{i=k-p+1}^{k} obj_i \quad \text{(financial form)}$$

In the second form, by passing the Boolean value **true** for argument `b`, the mean is taken from an equal number of values on either side of `k`, including `k`. Thus `p` must be an odd number:

$$\frac{1}{p} \sum_{i=k-p\backslash 2}^{k+p\backslash 2} obj_i \quad \text{(scientific form)}$$

It returns undefined, if either the left or right end of the sublist to be evaluated is not part of `obj`. The function does not accept structures including the value **undefined**.

By dividing each element before summation, the function avoids arithmetic overflows and also uses Kahan-Babuška summation to prevent round-off errors during summation.

stats.gsma is the iterator version of this function which traverses large distributions much faster.

See also: **stats.amean**, **stats.gsma**, **stats.gsmm**, **stats.smm**.

```
stats.smallest (obj [, k])
```

Returns the `k`-th smallest element in the numeric table, sequence or numarray `obj`. If `k` is not given, it is set to 1.

```
stats.smm (obj, k, p)
stats.smm (obj, k, p, b)
```

In the first form, computes the simple moving median of a table or sequence `obj` by sorting the last `p` numbers from the structure (`p` is also known as the `period`) including sample `k`, and then taking its median.

In the second form, by passing the Boolean value **true** for argument `b`, the simple moving median is determined by sorting an equal number of values on either side of `k`, including `k`, and then taking the median. Thus `p` must be an odd number.

The function is more robust than **stats.sma** to outliers in a period.

It returns undefined, if either the left or right end of the sublist to be evaluated is not part of `obj`. The function does not accept structures including the value **undefined**.

The function automatically non-destructively sorts the distribution `obj` if it is unsorted.

stats.gsmm is the iterator version of this function which traverses large distributions much faster.

See also: **stats.amean**, **stats.gsma**, **stats.gsmm**, **stats.sma**.

`stats.sorted (obj [, true] [, options])`

Sorts the table or sequence `obj` of numbers in ascending order and non-destructively up to and around twice as fast as **sort** if the structure contains (around) more than seven elements. It also ignores **undefined**'s. The type of return is defined by the type of the input.

If an element in `obj` is not a number, it is replaced with the number 0 before sorting.

By default, the function internally uses a recursive implementation of the Quicksort algorithm combined with a fallback to Heapsort in ill-conditioned situations, called Introsort.

You may exclusively use an iterative variant of the Quicksort algorithm by passing the second argument **true** or the string `'pixelsort'`, which may be faster on some older systems, especially with elements in completely random or in (nearly) ascending order. If the option `'nrquicksort'` is given, an alternative non-recursive algorithm described by Niklaus Wirth is being used. If the option `'heapsort'` is passed, the function uses the Heapsort algorithm. If the option `'quicksort'` is given, a traditional recursive Quicksort algorithm is being used.

See also: **sort**, **sorted**, **skycrane.sorted**, **stats.issorted**, **numarray.sort**.

`stats.spread (obj)`

Computes the population spread, i.e. the variance, of a distribution `obj` of numbers, and returns a number. The result is equal to:

$$\frac{1}{n} \sum_{i=1}^{n} obj_i^2 - \frac{1}{n^2} \left(\sum_{i=1}^{n} obj_i \right)^2$$

The function is around 10 percent faster than **stats.var** but is more susceptible to numeric overflows if the magnitudes of the observations are very large.

The function is implemented in Agena and included in the lib/stats.agn file.

`stats.standardise (obj [, option])`

Standardises a distribution by subtracting the arithmetic mean μ from each observation and then dividing by the population standard deviation (default) σ of the distribution:

$$\text{obj}_i \longrightarrow \frac{obj_i - \mu}{\sigma}$$

Depending on the type of its argument `obj`, the return is either a new table or sequence of the respective quotients, preserving the original order of the observations. You may alternatively divide by the sample standard deviation by passing the optional value **true** as the second argument. `obj` may be a table, sequence or numarray.

`stats.studentst (x [, nu])`

The Student's t-distribution has the probability density function:

$$\Gamma(\,(\text{nu}+1)/2\,)\,/\,\Gamma(\text{nu}/2)\,/\,\sqrt{\text{nu}^*\pi}\,/\,(1+t^2/\text{nu})\,\hat{}\,((\text{nu}+1)/2),$$

with `nu` a positive integer.

See also: **stats.cauchy**, **stats.chisquare**, **stats.fratio**, **stats.normald**.

`stats.sum (obj)`

`stats.sum (f, obj [, ···])`

The function has been deprecated, please use **stats.sumdata** instead.

`stats.sumdata ([f, ] obj [, p [, x`$_m$` [, ···]]])`

Sums up all the powers `p` of the given table, sequence or numarray `obj` of n elements about the origin x_m and returns a number. It is equivalent to:

$$\sum_{i\,=1}^{n}\left(\text{obj}_i - x_m\right)^p$$

If only `obj` is given, the power `p` defaults to 1, and the origin x_m defaults to 0. If given, `p` and x_m must be numbers. If `obj` is empty, the function returns **fail**.

If a function `f` is given, it only sums up the values in `obj` satisfying `f`, which should return a Boolean. If `f` has more than one argument, then its second to last argument must be given right after x_m.

Examples:

```
> import stats;

> stats.sumdata(<< x -> x > 2 >>, seq(1, 2, 3, 4)):
7

> stats.sumdata(<< x, y -> x + y > 2 >>, seq(1, 2, 3, 4), 1, 0, 1):
9
```

The function uses Kahan-Babuška round-off error prevention.

See also: **math.koadd**, **sumup**, **stats.cumsum**, **stats.fsum**, **stats.moment**, **stats.sumdataln**.

stats.sumdataln ([f,] obj [, p [, x_m [, ···]]])

Sums up all the natural logarithms of the powers `p` of the given table, sequence or numarray `obj` of n elements about the origin x_m and returns a number. It is equivalent to:

$$\sum_{i=1}^{n} \ln\left(\left(obj_i - x_m\right)^p\right)$$

If only `obj` is given, the power `p` defaults to 1, and the origin x_m defaults to 0. If given, `p` and x_m must be numbers. If `obj` is empty, the function returns **fail**.

If a function `f` is given, it only sums up the values in `obj` satisfying `f`, which should return a Boolean. If `f` has more than one argument, then its second to last argument must be given right after x_m. For examples, please see **stats.sumdata**.

stats.tovals (obj)

Converts all string values in the structure `obj` to Agena numbers or complex numbers and returns a new structure. The type of return is determined by the type of `obj`.

stats.trimean (obj [, p])

If `p` is not given, the function determines the 1st quartile Q1 and the 3rd quartile Q3 along with the median Q2 of a distribution `obj` and returns the trimean (Q1 + 2*Q2 + Q3)/4 along with the median. `obj` may be a table, sequence or numarray.

If `p`, an integer in the range [0 .. 100) is given, instead of the first and third quartiles the `p`-th and 100 - `p`-th percentile ranks are the lower and upper margins in the computation.

When compared to the median, the trimean is a means to determine whether a distribution is biased in its first or second half. If the distribution is not sorted, it

automatically sorts it non-destructively, where any non-numeric elements are set to 0.

See also: **stats.iqr**, **stats.trimmean**, **stats.winsor**.

`stats.trimmean (obj, f)`

Returns the arithmetic mean of the interior of a distribution `obj` (a table, sequence or numarray), where the number `f` ∈ [0, 1) determines the fraction of the data that is to be excluded from the margins.

The number p of data to be excluded from `obj` is always rounded down to the nearest even number. The function then does not take into account p/2 points from the left margin and p/2 points from the right margin when calculating the average using Kahan-Babuška round-off error prevention. The function does not sort the distribution.

The return is a number. It returns **fail**, if the distribution includes less than two elements.

The function is implemented in Agena and included in the lib/stats.agn file.

See also: **stats.amean**, **stats.winsor**.

`stats.var (obj [, sample [, option]])`

Returns the variance of all numeric values in table, sequence or numarray `obj` as a number. If `obj` is a table, it is assumed to be an array, non-positive integral keys (including strings, etc.) are ignored.

If `sample` is not given or does not evaluate to **true,** the population variations will be returned, where μ is the arithmetic mean of a distribution:

$$\sigma^2 = \frac{1}{n} \sum_{i=1}^{n} \left(obj_i - \mu \right)^2$$

If `sample` is given and is **true**, the (unbiased) sample variance will be returned:

$$\sigma^2 = \frac{1}{n-1} \sum_{i=1}^{n} \left(obj_i - \mu \right)^2$$

If `option` of any type is passed, the variation coefficient $\sigma^2 / |\mu|$ is determined to make different distributions comparable.

The function returns **fail** if `obj` contains less than two elements.

The function is implemented in Agena and included in the lib/stats.agn file.

See also: **stats.ad**, **stats.ios**, **stats.mad**, **stats.sd**, **stats.spread**.

`stats.weights (obj)`

`stats.weights (x [, ···])`

In the first form, the function inserts all the elements in sequence `obj` into a new sequence and returns it.

In the second form, it inserts all the given arguments into a new sequence and returns it.

In both forms, only numbers and pairs are accepted in the sequence or argument list. In case of a pair x:n, x denotes an observation and n, a non-negative integer, denotes the number of occurrence of x, so x is inserted n times into the new sequence.

`stats.winsor (obj [, n])`

Returns the winsorised mean of all numeric values in table or sequence `obj`.

The function first replaces the first `n` percent of a distribution at the low and high end with the most extreme remaining values, and then calculates the arithmetic mean of the entire modified distribution. By default, `n` is 10.

The winsorised mean is more resistant to outliers than the traditional arithmetic mean.

See also: **stats.trimean**, **stats.trimmean**.

`stats.zscore (obj)`

Returns a univariate function `z(x)` computing the z-score (standard score) of a sample x in the table, sequence or numarray `obj` - the number of standard deviations x is above or below the mean according to the formula: $z(x) = (x - \mu)/\delta$, where μ denotes the arithmetic mean of `obj`, and δ its standard deviation.

The resulting function returns a positive number if x is above the mean and a negative number if it is below. It does, however, not check whether x is part of `obj`. The result is computed using Kahan-Babuška round-off error prevention for μ and δ.

The function is implemented in Agena and included in the lib/stats.agn file.

11.15 long - 80-Bit Floating-Point Arithmetic

This package implements 80-bit floating-point numbers and arithmetic. It is at least 80 times faster than the **mapm** package with the same precision.

Note that 80-bit floating-point arithmetic provided by this new package is still at least 12 times slower than Agena's built-in 64-bit arithmetic so it is useful only if you need extended precision with 19 significant digits and a range of approximately 3.65×10^{-4951} to 1.18×10^{4932}.

As a *plus* package, it is not part of the standard distribution and must be activated with the **import** statement, e.g. `import long`. Example:

```
> import long;

> a, b := long.double(1), long.double(2)

> a + b:
longdouble(3.0000000000000000000)

> exp(a):
longdouble(2.7182818284590452354)

> long.tonumber(ans):  # convert back to Agena number (64-bit)
2.718281828459
```

Note that relational operators such like = or <> cannot compare longdoubles with numbers as they are different types, and the result would always be **false**. Arithmetic binary operators, however, such like +, **%**, however, can process a mix of numbers and longdoubles and return a longdouble as the result. Also, most unary operators and functions with one or two arguments can process both longdoubles and ordinary Agena numbers.

Creation, Conversion, Iteration:

long.double (x)

Creates a longdouble from the ordinary Agena number x, returning a userdata of type longdouble with the longdouble metatable attached.

x may also be a string representing a number - with constants, you might prefer this to avoid round-off errors. For a predefined set of constants, also check the end of this subchapter.

long.tonumber (x)

Converts the longdouble x into an ordinary Agena number, losing precision.

`long.tostring (x [, format])`

Converts the longdouble `x` into a string with 19 fractional digits by default. You can pass a string `format` that includes the `ld` specifier to control the output, e.g.

```
> long.tostring(long.Pi, "%.15ld"):
```

returns

```
3.141592653589793
```

`long.count ([start [, step [, stop]]])`

Returns an iterator function that, each time it is called, returns a new longdouble.

If no argument is given, the first number returned by the iterator is 0, the next call returns 1, the next one 2, and so forth. This means that the longdouble returned with each call is increased by 1.

If only `start` is given, the first value returned by the iterator is `start`, the next call returns `start + 1`, the next one `start + 2`, and so forth. This means that the longdouble returned with each call is increased by 1.

If `start` and `step` are given, the first value returned by the iterator is `start`, the next call returns `start + step`, the next one start + 2*`step`, and so forth. This means that the number returned with each call is increased by `step`, which may be negative. In the latter case the next value returned will be less than the current returned value.

If `stop` is given, the iterator returns **null** if the counter value exceeds `stop`. Default is +**long.infinity**.

If `start` or `step` are not longdoubles or numbers, the factory issues an error.

If `start` or `step` are non-integral, the function applies Neumaier summation to avoid round-off errors.

Example:

```
> import long;

> f := long.count(long.double(1), long.double(-0.1), long.double(-1));

> while c := f() do print(c) od; # counts down
```

A note in advance: All the functions and most of the operators are also available as functions which names always start with "long.x". So, for example, the **sin** operator is also available as the function **long.xsin**, and the **math.chop** function has the alias **math.xchop**.

Basic Arithmetic Operations:

x ± y

The operator computes $x + y$, i.e. performs an addition.

x − y

The operator computes $x - y$, i.e. performs a subtraction.

x * y

The operator computes $x * y$, i.e. performs a multiplication.

x ∕ y

The operator computes x / y, i.e. performs a division.

x ∖ y

The operator computes $x \setminus y$, i.e. performs an integer division.

x % y

The operator computes $x \% y$, i.e. returns the modulus.

x ˆ y

The operator computes $x \hat{} \ y$, i.e. raises x to the power of y, where y represent any arithmetic number.

x ** y

The operator computes $x ** y$, i.e. raises x to the power of y, where y represent an integer.

recip (x)

The operator returns the inverse $1/x$.

abs (x)

The operator will return the absolute value of x.

sign (x)

Determines the sign of x. The result of the operator is determined as follows:
- 1, if x > 0,
- -1, if x < 0,
- **undefined**, if x = **undefined**,
- 0 otherwise, even for -0.

signum (x)

Determines the sign of x. The result of the operator is determined as follows:

- 1, if $x \geq 0$
- -1 otherwise.

long.copysign (x, y)

Returns a longdouble with the magnitude of x and the sign of y, i.e. abs(x) * sign(y). If y is 0, then its sign is considered to be 1.

long.exponent (x)

Returns the exponent e of x such that **long**.**mantissa**(x) * 2^e equals x.

long.fma (x, y, z)

Performs the fused multiply-add operation `(x * y) + z`, with the intermediate result *not* rounded to the destination type, to improve the precision of a calculation.

long.fmod (x, y)

Computes the remainder from the division of numerator x by denominator y. The return value is x - n * y, where n is the quotient of x divided by y, rounded towards zero to an integer.

long.hypot (x, y)

Returns $\sqrt{x^2 + y^2}$. It is the length of the hypotenuse of a right triangle with sides of length x and y, or the distance of the point (x, y) from the origin. The function is slower but more precise than using **sqrt** along with **square**, avoiding over- and underflow.

long.hypot2 (x)

Returns $\sqrt{1 + x^2}$, avoiding over- and underflow.

long.hypot3 (x)

Returns $\sqrt{1 - x^2}$, avoiding over- and underflow.

`long.hypot4 (x)`

Returns $\sqrt{x^2 - y^2}$, avoiding over- and underflow.

`long.koadd (x, y [, q])`

The function adds `x` and `y` using Kahan-Ozawa round-off error prevention and returns two longdoubles: the sum of `x` and `y` plus the updated value of the correction variable `q`. The optional correction variable `q` should be 0 at first invocation, and the previously returned correction variable otherwise - if `q` is not given, it defaults to 0.

A typical usage should look like:

```
> x, q -> long.double(0);
> y := long.double('0.1');
> while x < long.double(1) do
>    x, q := long.koadd(x, y, q)
> od;
> print(s + q);
```

`long.fdim (x, y)`

Computes `x` - `y` if `x` > `y`, and return 0 otherwise.

`long.mantissa (x)`

Returns the mantissa m of `x` such that m * 2^**long.exponent**(`x`) equals `x`. See also: **long.significand**.

`long.modf (x)`

Returns the integral part of `x` and its fractional part. The integral part is rounded towards zero. Both the integral and fractional part of the return have the same sign as `x`. The sum of the two values returned equals `x`.

`long.pytha (a, b)`

Computes the Pythagorean equation $c^2 = a^2 + b^2$, without undue underflow or overflow, for longdoubles or numbers `a, b`.

`long.pytha4 (a, b)`

Computes a^2 - b^2, without undue underflow or overflow, for longdoubles or numbers `a, b`.

`long.significand (x)`

Returns the mantissa of x in a normalised form, in the range[1, 2), with **long.significand**(x) = 2***long.mantissa**(x) = **long.ldexp**$(x,$ -**long.ilog2**$(x))$. If x is 0, the return is 0.

Relations:

A note in advance: The 80-bit infinity representation is represented by the constant **long.infinity** and not **infinity**, and that of "undefined" is **long.undefined**.

`x = y`

The binary operator returns **true** if x is exactly equal to y, and **false** otherwise.

`x <> y`

The binary operator returns **true** if x is not exactly equal to y, and **false** otherwise.

`x ~= y`

The binary operator returns **true** if x is approximately equal to y, and **false** otherwise. See also **long.approx**.

`x ~<> y`

The binary operator returns **false** if x is approximately equal to y, and trueotherwise.

`x < y`

The binary operator returns **true** if x is less than y, and **false** otherwise.

`x <= y`

The binary operator returns **true** if x is less than or equal to y, and **false** otherwise.

`x > y`

The binary operator returns **true** if x is greater than y, and **false** otherwise.

`x >= y`

The binary operator returns **true** if x is greater than or equal to y, and **false** otherwise.

`long.isequal (x, y)`

Compares x with y and returns **true** if $x = y$, and **false** otherwise.

`long.isunequal (x, y)`

Compares x with y and returns **true** if $x <> y$, and **false** otherwise.

`long.isless (x, y)`

Compares x with y and returns **true** if $x < y$, and **false** otherwise.

`long.islessequal (x, y)`

Compares x with y and returns **true** if $x <= y$, and **false** otherwise.

`long.approx (x, y [, eps])`

Compares the x and y and checks whether they are approximately equal. If `eps` is omitted, **Eps** is used.

The algorithm uses a combination of simple distance measurement (`|x-y|` $\leq$ `eps`) suited for values `near` 0 and a simplified relative approximation algorithm developed by Donald H. Knuth suited for larger values (`|x-y|` $\leq$ `eps` `* max(|x|,` `|y|)`), that checks whether the relative error is bound to a given tolerance `eps`.

The function returns **true** if x and y are considered equal or **false** otherwise. If both a and b represent **infinity**, the function returns **true**. The same applies to a and b being -**infinity** or **undefined**.

`long.fmax (x, y)`

Returns x if $x > y$, and y otherwise.

`long.fmin (x, y)`

Returns x if $x < y$, and y otherwise.

Powers and Roots:

`square (x)`

The operator squares x and returns x**2.

`cube (x)`

The operator raises x to the power of 3.

`x squareadd c`

The operator computes $x \hat{} 2 + c$, preventing round-off errors. See also **long.fma**.

`sqrt (x)`

Returns the square root of x.

If x is a number and negative, the operator returns **undefined**.

long.cbrt (x)

Returns the cubic root of the number x.

long.root (x [, n])

Returns the non-principal n-th root of x. n must be an integer and is 2 by default. Note that since the function computes the non-principal root.

invsqrt (x)

The operator computes the inverse square root x, i.e. 1/**sqrt**(x).

Exponentiation & Logarithms:

antilog2 (z)

The operator computes 2^z, i.e. 2 raised to the power of z.

antilog10 (z)

The operator computes 10^z, i.e. 10 raised to the power z.

exp (x)

Exponential function; the operator returns the value e^x, with e Euler's number.

long.expminusone (x)

Returns a value equivalent to exp(x) - 1. It is computed in a way that is accurate even if x is near 0, since exp($\sim$0) and 1 are nearly equal.

ln (x)

Natural logarithm of x with the base e^1. If x is non-positive, the operator returns **undefined**.

long.lnplusone (x)

Returns a value equivalent to ln(1 + x). It is computed in a way that is accurate even if x is near zero.

log (x, b)

The operator returns the logarithm of x to the base b.

`long.log2 (x)`

Returns the base-2 logarithm of x.

`long.ilog2 (x)`

Returns the integral base-2 logarithm of x.

`long.log10 (x)`

Returns the base-10 logarithm of x.

Trigonometric Functions and Operators:

`sin (x)`

The operator returns the sine of x (in radians).

`cos (x)`

The operator returns the cosine of x (in radians).

`tan (x)`

The operator returns the tangent of x (in radians).

`long.csc (x)`

The function returns the secant of x (in radians), i.e. 1/**sin**(x).

`long.sec (x)`

The function returns the secant of x (in radians), i.e. 1/**cos**(x).

`long.cot (x)`

The function returns the cotangent of x (in radians), i.e. 1/**tan**(x).

`sinc (x)`

The operator returns the un-normalised cardinal sine of x (in radians), i.e. **sin**(x)/x, with **sinc**(**long.double**(0)) = **long.double**(1).

Hyperbolic Trigonometric Functions and Operators:

sinh (x)

The operator returns the hyperbolic sine of x (in radians).

cosh (x)

The operator returns the hyperbolic cosine of x (in radians).

tanh (x)

The operator returns the hyperbolic tangent of x (in radians).

Inverse and Inverse Hyperbolic Trigonometric Functions and Operators:

arccos (x)

The operator returns the inverse cosine operator (x in radians).

long.arccosh (x)

The function computes the inverse hyperbolic cosine of x (in radians).

arcsin (x)

The operator computes the inverse sine operator (in radians).

long.arcsinh (x)

The function returns the inverse hyperbolic sine of x (in radians).

arctan (x)

The operator computes the inverse tangent operator (in radians).

long.arctanh (x)

The function returns the inverse hyperbolic tangent of x (in radians).

long.arccsc (x)

The function returns the inverse cosecant of x (in radians).

arcsec (x)

The operator returns the inverse secant of x (in radians).

long.arccot (x)

The function returns the inverse cotangent of x (in radians).

Error Functions:

long.erf (x)

Returns the error function of number x. It is defined by $\mathrm{erf}(x) = \dfrac{2}{\sqrt{\pi}} \displaystyle\int_{t=0}^{x} e^{-t^{\wedge}2}$.

long.erfc (x)

Returns the complementary error function of x. It is defined by $\mathrm{erfc}(x) = 1 - \mathrm{erf}(x)$.

Rounding & Related:

long.ceil (x)

Rounds upwards to the nearest integer larger than or equal to x.

entier (x)

The operator rounds x downwards to the nearest integer. Same as:

long.floor (x)

Rounds x downwards to the nearest integer. Also: long.floor(x) = ceil(x) = -entier(-x).

int (x)

The operator rounds x to the nearest integer towards zero.

frac (x)

The operator returns the fractional part of x, i.e. x – **int**(x), preserving the sign.

long.round (x [, d])

Rounds x to its d-th digit, using the round-half-up method. If d is omitted, the longdouble is rounded to the nearest integer. If d is positive, the function rounds to the d-th fractional digit. If d is negative, it rounds to the d-th integral digit. **long.round** treats positive and negative values symmetrically, and is therefore free of sign bias.

long.chop (x [, eps [, method [, n]]]])

Shrinks x more or less near zero to exactly zero, using one of several methods, passed as an integer. The default for `eps` is **DoubleEps**. The standard `method` is 0 for hard shrinking. Integral `n` is used in the SmoothGarrote `method`.

method	Comment	Value	Domain
0	"Hard", performs hard shrinking	0 x	$\lvert x \rvert \leq$ eps $\lvert x \rvert >$ eps
1	"Soft", performs soft shrinking	0 $\mathrm{sign}(x)\,(\lvert x \rvert - \mathrm{eps})$	$\lvert x \rvert \leq$ eps $\lvert x \rvert >$ eps
3	"PiecewiseGarrote"	0 $x - \mathrm{eps}^2/x$	$\lvert x \rvert \leq$ eps $\lvert x \rvert >$ eps
4	"SmoothGarrote"; with $n \to \infty$, goes to "Hard" shrinking	$x^{2n+1}/(x^{2n} + \mathrm{eps}^{2n})$	any x
5	"Hyperbola"	0 $\mathrm{sign}(x)\,\sqrt{x^2 - \mathrm{eps}^2}$	$\lvert x \rvert \leq$ eps $\lvert x \rvert >$ eps

Method 2 has not been implemented.

Checks:

nonzero (x)

Checks whether x is not 0. The operator returns **true** or **false**.

zero (x)

Checks whether x is 0, respectively. The operator returns **true** or **false**.

even (x)

Checks whether x is even. The operator returns **true** if x is even, and **false** otherwise. With non-integral numbers, the operator returns **false**.

odd (x)

The operator checks whether x is odd. The operator returns **true** if x is odd, and **false** otherwise. With non-integral longdoubles, the operator returns **false**.

float (x)

Checks whether x has a fraction, i.e. not integral, and returns **true** or **false**.

With +/-**infinity** and **undefined**, returns **false**.

integral (x)

Checks whether x is integral, i.e. does not contain a fraction, and returns **true** or **false**.

With +/-**infinity** and **undefined**, returns **false**.

`finite (x)`

The operator checks whether x is neither ±**infinity** nor **undefined**. The operator returns **true** or **false**.

`nan (x)`

Checks whether x evaluates to **undefined**. The operator returns **true** or **false**.

Miscellaneous:

`long.eps ([x [, option]])`

The function returns the machine epsilon, the relative spacing between $|x|$ and its next larger longdouble in the machine's floating point system. If no argument is given, x is set to 1.

When given any second argument, the function computes a `mathematical` epsilon value that is also dependent on the magnitude of its argument x. It can be used in difference quotients, etc., for it prevents huge precision errors with computations on very small or very large numbers. The mathematical epsilon with respect to x is equal to x* **sqrt**(**long.eps**(x)).

`long.gsolve (A)`
`long.gsolve (A, v)`

Performs Gaussian elimination on a system of linear equations.

In the first form, `A` must be an augmented m x n matrix created with **linalg.matrix** with m+1 = n. In the second form, `A` is a square matrix and `v` a right-hand side vector, created with **linalg.vector**.

linalg.matrix and **linalg.vector** accept standard Agena numbers only, so the input is 64-bit precision but the output is 80-bit.

The return is the solution vector with longdoubles. It returns **infinity** if an infinite number of solutions has been found, and **undefined** if no solutions exists. It returns **fail** if it could not determine whether no or an infinite number of solutions exist.

Example:

```
> import linalg, long

> A := linalg.matrix([1, 1, 1, 2], [6, -4, 5, 31], [5, 2, 2, 13]):
[ 1,  1, 1,  2 ]
[ 6, -4, 5, 31 ]
[ 5,  2, 2, 13 ]

> long.gsolve(A):
[longdouble(3.000000..), longdouble(-2.000000..), longdouble(0.999999..)]
```

`long.nextafter (x, y)`

Returns the next machine floating-point number of x in the direction toward y.

`long.norm (x, a1:a2 [, b1:b2])`

Converts x in the scale [$a1$, $a2$] to one in the scale [$b1$, $b2$]. The second and third arguments must be pairs of longdoubles. If the third argument is missing, then x is converted to a longdouble in [0, 1].

`long.rempio2 (x [, option])`

Conducts an argument reduction of x into the range $|y| < \dfrac{\pi}{2}$ and returns y = x - N*$\dfrac{\pi}{2}$. If any `option` is given, then the function also returns N, or actually the last three digits of N. The number of operations conducted are independent of the exponent of the input.

`long.unm (x)`

Negates a number or longdouble x and returns it, same as the expression -x.

`long.wrap (x [, a [, b]])`

Conducts a range reduction x to the interval [a, b]. If $x \in$ [a, b), x is simply returned.

In the second form, if a is not given, a is set to $-\pi$ and b to $+\pi$. If a is given but not b, a is set to -a and b to +a, so a should be positive.

`long.signbit (x)`

Checks whether x has its sign bit set and returns **true** or **false**. For example, although -0 = 0, **long.signbit**(-0) $\Rightarrow$ **true** and **long.signbit**(0) $\Rightarrow$ **false**.

`long.frexp (x)`

Returns the mantissa m and the exponent e of x such that $x = m2^e$. e should be integral, and the value of m is in the range [0.5, 1) (or zero when x is zero). The operation is bijective, i.e. **long.ldexp**(**long.frexp**(x)) = x.

`long.ldexp (m, e)`

Returns $m2^e$ (e should be integral).

`long.multiple (x, y [, option])`

Checks whether x is a multiple of y, i.e. whether x/y evaluates to an integral, and returns **true** or **false**.

Also returns **true** with $x = 0$ and any non-zero y.

If y is zero, **long.undefined** or $+/-$**long.infinity**, the function returns **fail**.

By passing the optional third argument **true**, a tolerant check is done, with subnormal x or y first converted to zero, and a subsequent approximate equality check to the nearest integer of x/y. The tolerance value internally used is the value of **DoubleEps** at the time of the function call.

In most cases, it may suffice to just call **integral**(x/y).

IEEE:

`long.fpclassify (x)`

For the given x, returns

- **long.fp_nan** if x is **undefined**,
- **long.fp_infinite** if x is infinite, i.e. $+/-$**infinity**,
- **long.fp_subnormal** if x is subnormal,
- **long.fp_zero** if x is zero,
- **long.fp_normal** if x is normal, including irregular values $\geq 2^{52}$.

`long.isundefined (x)`

Returns **true** if **long.fpclassify**(x) = **long.fp_nan**, and **false** otherwise.

`long.isinfinite (x)`

Returns **true** if **long.fpclassify**(x) = **long.fp_infinite**, and **false** otherwise.

`long.iszero (x)`

Returns **true** if **long.fpclassify**(x) = **long.fp_zero**, and **false** otherwise.

`long.isnormal (x)`

Returns **true** if **long.fpclassify**(x) = **long.fp_normal**, and **false** otherwise.

`long.issubnormal (x)`

Returns **true** if **long.fpclassify**(x) = **long.fp_subnormal**, and **false** otherwise.

`long.isfinite (x)`

Returns **true** if **long.fpclassify**(x) $<>$ **long.fp_nan** and **long.fpclassify**(x) $<>$ **long.fp_infinite**, and **false** otherwise.

Available constants are:

Constant	Value	Comment
long.Pi	π	
long.Pi2	2π	
long.PiO2	$\pi/2$	
long.PiO4	$\pi/4$	
long.PiO180	$\pi/180$	radians per degree
long.InvPi2	$1/(2\pi)$	
long.InvPiO4	$4/\pi$	
long.InvPiSqO4	$4//\pi^2$	
long.E	$E = \exp(1)$	
long.sqrt2	$\sqrt{2}$	
long.sqrt3	$\sqrt{3}$	
long.ln2	$\ln(2)$	
long.Invln2	$1/\ln(2)$	
long.Invsqrt2	$1/\sqrt{2}$	
long.Phi	$(1 + \sqrt{5})/2$	Golden Ratio
long.InvPhi	$1/((1 + \sqrt{5})/2)$	inverse Golden Ratio
long.InvPhiSq	$(1/((1 + \sqrt{5})/2))^2$	
long.lnPhi	$\ln(1 + \sqrt{5})/2$	logarithm of Golden Ratio
long.InvlnPhi	$1/\ln(1 + \sqrt{5})/2$	inverse
long.naught long.nought	0	zero
long.one	1	
long.two	2	
long.three	3	
long.four	4	
long.five	5	
long.six	6	
long.seven	7	
long.eight	8	
long.nine	9	
long.ten	10	
long.eleven	11	
long.twelve	12	
long.fifty	50	
long.hundred	100	
long.thousand	1,000	
long.half	0.5	
long.quarter	0.25	
long.tenth	0.1	
long.fifth	0.2	
long.hundredth	0.01	
long.thousandth	0.001	
long.threequarter	0.75	
long.third	1/3	

Constant	Value	Comment
long.sixth	1/6	
long.eighth	0.125	
long.twelfth	1/12	
long.sixteenth	0.0625	
long.infinity	infinity	
long.undefined	undefined	

These constants have all been defined in source file lib/long.agn.

Chapter **Twelve**

Input & Output

12 Input & Output

12.1 io - Input and Output Facilities

The I/O library provides two ways for file manipulation.

Summary of functions:

Opening and closing files:

io.open, **io.close**.

Reading data:

io.input, **io.lines**, **io.read**, **io.readfile**, **io.readlines**.

Writing data:

io.output, **io.write**, **io.writefile**, **io.writelines**.

File positions:

io.eof, **io.filepos**, **io.move**, **io.seek**, **io.skiplines**.

File locking:

io.lock, **io.unlock**.

File buffering:

io.setvbuf, **io.sync**

Interaction with applications:

io.pcall, **io.popen**, **io.close**

Keyboard interaction:

io.anykey, **io.getkey**, **io.kbdgetstatus**, **io.keystroke**

Windows clipboard interaction

io.getclip, **io.putclip**.

Miscellaneous:

io.clearerror, **io.ferror**, **io.fileno**, **io.filesize**, **io.isfdesc**, **io.mkstemp**, **io.nlines**, **io.isopen**, **io.tmpfile**, **io.truncate**.

Usage:

1. The first one uses *file handles*; that is, there are operations to set a default input file and a default output file, and all input/output operations are over these default files. File handles are values of type userdata and are used as in the following example:

   Open a file and store the file handle to the name `fh`:

   ```
   > fh := io.open('d:/agena/src/change.log'):
   file(7803A6F0)
   ```

   Read 10 characters:

   ```
   > io.read(fh, 10):
   Change Log
   ```

   Close the file:

   ```
   > io.close(fh):
   true
   ```

   In the following descriptions of the **io** functions, file handles are indicated with the argument `filehandle`.

   The table `io` provides three predefined file handles with their usual meanings from C: `io.stdin`, `io.stdout`, and `io.stderr`.

2. The second style uses file names passed as strings like `'d:/agena/lib/library.agn'`. File names are always indicated with the argument `filename` in this chapter.

Unless otherwise stated, all I/O functions return **null** on failure (plus an error message as a second result) and some value different from **null** on success.

`io.anykey ()`

Checks whether a key is being pressed and returns either **true** or **false**. A common usage is as follows:

```
> while io.anykey() = false do od; # wait until a key has been pressed
```

The function works in the OS/2, Solaris, Linux, DOS, and Windows editions only. On Lion, the function sometimes echoes the key being pressed. On other systems, it returns **fail**.

See also: **io.getkey**, **io.read**.

`io.clearerror (filehandle)`

Clears the end-of-file and error indicators for the file denoted by `filehandle`. The function returns nothing.

See also: **io.eof**, **io.ferror**.

`io.close ([filehandle, ···])`

Closes one or more files. Note that files are automatically closed when their handles are garbage collected, but that takes an unpredictable amount of time to happen.

Without a `filehandle`, closes the default output file.

The function also deletes the file handles and the corresponding filenames from the **io.openfiles** table if the files could be properly closed.

The function returns **true** on success and **false** otherwise. With pipes, also returns the exit code of the application run.

See also: **io.open**, **io.popen**.

`io.eof (filehandle)`

Checks whether the end of the file denoted by `filehandle` has been reached and returns **true** or **false**.

See also: **io.clearerror**, **io.ferror**.

`io.ferror (filehandle)`

Checks the error indicator for the file denoted by `filehandle` and returns **true** if set or **false** if not set.

See also: **io.clearerror**, **io.eof**.

`io.fileno (filehandle)`

Returns the file descriptor, an integer, associated with the stream referenced by `filehandle`, which is of type **userdata/file**. It is useful for informative purposes, only.

The return cannot be used as a substitute to `filehandle` in calls to **io** functions, and which require a handle of type **userdata/file**.

The function issues an error if `filehandle` is not of type **userdata/file** or if does not reference an open file.

See also: **io.isfdesc**.

`io.filepos (filehandle)`

Returns the current position in the file denoted by its file handle `filehandle`, and returns a non-negative number.

See also: **io.seek**.

`io.filesize (filehandle)`

Returns the size of an open file denoted by its file handle `filehandle` and returns the number of bytes as a non-negative integer.

`io.getclip ()`

Returns the contents of the Windows clipboard as a string. If the clipboard could not be accessed, it returns **fail** plus an error string. It also returns fail and an error string, if the clipboard contains a binary object.

The function is available in the Windows edition only.

See also: **io.putclip**.

`io.getkey ([anything])`

If no argument is given, waits until a key is pressed and returns its ASCII number. If any argument is passed, the function waits until a key is pressed, but returns nothing.

The function is available in the OS/2, Solaris, Linux, Mac OS X, DOS, and Windows editions only.

See also: **io.anykey**, **io.read**.

`io.infile (filename, pattern)`
`io.infile (filehandle, pattern)`

Checks whether the file given by the name `filename` or the file denoted by its descriptor `filehandle` includes a `pattern` of type string, and returns **true** or **false**. The function supports pattern matching.

See also: **io.readfile**, **utils.findfiles**.

io.input (filehandle)

io.input (filename)

io.input ()

When called with a file name, it opens the named file (in text mode), and sets its handle as the default input file. When called with a file handle, it simply sets this file handle as the default input file. When called without parameters, it returns the current default input file.

In case of errors this function raises the error, instead of returning an error code.

io.isfdesc (filehandle)

Checks whether `filehandle` is a valid file handle. Returns **true** if `filehandle` is an open file handle, or **false** if `filehandle` is not a file handle.

See also: **io.fileno**, **io.isopen**.

io.isopen (filehandle)

Checks whether `filehandle` references an open file. Returns **true** if `filehandle` is an open file handle, or **false** if `filehandle` is not a file handle. Thus it also returns **false** if `filehandle` is not of type **userdata/file**. Contrary to **io.isfdesc**, it also detects invalid file positions caused by files too large or if the stream referenced by `filehandle` does not support file positioning.

Please note that the function cannot detect whether a file has been opened by another application.

The function is five times slower than **io.fdesc**.

See also: **io.fileno**, **io.isfdesc**.

io.kbdgetstatus ()

OS/2 only: Get status information about the keyboard. The function returns a table with the contents of the KBDINFO structure after the call to the C API function KbdGetStatus. See http://www.edm2.com/index.php/KbdSetStatus_(FAPI) for the meaning of the results.

io.keystroke (c)

Windows only: emulates a keystroke for the given ASCII value (an integer) `c` and dumps the character representing `c` to the currently active window. For security, newlines, carriage returns, and CTRL-Z's will not be accepted as input.

```
io.lines (filename)
io.lines (filehandle)
io.lines ()
```

In the first form, the function opens the given file denoted by string `filename` in read mode and returns an iterator function that, each time it is called, returns a new line from the file.

In the second form, the function opens the given file in read mode and returns an iterator function that, each time it is called, returns a new line from the file.

Therefore, the construction

```
    for keys line in io.lines(f) do body od
```

will iterate over all lines of the file denoted by `f`, where `f` is either a file name or file handle. When the iterator function detects the end of file, it returns **null** (to finish the loop) and automatically closes the file if a filename is given. In case of a file handle, the file is not closed.

The call `io.lines()` (without a file name) iterates over the lines of the default input file. In this case it does not close the file when the loop ends.

See also: **io.readlines**.


```
io.lock (filehandle)
io.lock (filehandle, size)
```

The function locks the file given by its handle `filehandle` so that it cannot be read or overwritten by other applications.

In the first form, the entire file is locked in UNIX-based systems. In Windows, only 2^{63} bytes are locked, so you have to use the second form described below in Windows after the file has become larger than 2^{63} bytes ($=$ 8,589,934,592 GBytes).

In the second form the function locks `size` bytes from the current file position. Locked blocks in a file may not overlap. `size` may be larger than the current file length.

The function returns **true** on a successful lock, and **false** otherwise.

Note that other applications that do not use the locking protocol may nevertheless have read and write access to the file.

See also: **io.unlock**.

`io.maxopenfiles ([n])`

In Solaris, Linux Mac OS X and Windows, returns the maximum number of open files, minus 2 for stdin & stdout, if no argument is given - or sets the maximum number of files that are allowed to be opened simultaneously if an integer `n` is given.

On OS/2 and DOS, returns the maximum number of open files, but you cannot change this number.

On the platforms given above, when `n` is not given, returns **undefined** in case of errors. On all other platforms, the function always returns **undefined**.

`io.mkstemp (template)`

The function creates a unique temporary filename from the given `template`, a string, which must always end with six (capital) `x`'s and returns it as a string. It also creates a file of the same name, but does not open it.

Example: `io.mkstemp('fileXXXXXX')`.

See also: **io.tmpfile**, **os.remove**, **os.tmpdir**, **os.tmpname**.

`io.move (filehandle, n)`

Moves the current file position of the open file denoted by its `filehandle` either to the left or the right.

If `n` is a positive integer, then the file position is moved `n` characters to the right, if it is a negative integer, it is moved `n` characters to the left. If `n` is zero, the position is not changed at all.

The function returns **true** on success and **false** otherwise.

See also: **io.seek**.

`io.nlines (filename)`
`io.nlines (filehandle)`

The function counts the number of lines in the (text) file denoted by `filename` or `filehandle` and returns a non-negative integer.

See also: **io.skiplines**.

`io.open (filename [, mode])`

This function opens a file, given by the string `filename`, in the mode specified in the string `mode`. It returns a new file handle of type **userdata/file**. The function does not lock the file (see **io.lock**).

The function also enters the newly opened file into the **io.openfiles** table in the following format: [filehandle ~ [filename, mode]].

In case of errors, the function quits with an error.

The `mode` string can be any of the following:

- **'r'**, **'read'**: read mode (the default);
- **'w'**, **'write'**: write mode only; if the file already exists, it is truncated to zero length;
- **'a'**, **'append'**: append mode;
- **'r+'**: update mode (both reading and writing), all previous data is preserved; the initial file position is at the beginning of the file;
- **'w+'**: update mode (reading and writing), all previous data is erased;
- **'a+'**: append update mode (reading and appending), previous data is preserved, writing is only allowed at the end of file.

The `mode` string may also have a `'b'` at the end, which is needed in some systems to open the file in binary mode. This string is exactly what is used in the standard C function `fopen`.

See also: **io.close**, **io.lock**.

io.output ([filehandle])

Similar to **io.input** but operates over the default output file.

io.pcall (prog [, mode])

Starts programme `prog` (passed as a string) in a separated process, sends and receives data to this programme via stdout - if `mode` is `'r'`, or `mode` is not given -, or writes data to this programme if `mode` is `'w'`. After communication finishes, the connection is automatically closed.

The return is a sequence of strings containing the result sent back by the application.

The function thus is a combination of **io.popen**, **io.readlines**, and **io.pclose**, has been written in Agena, and is included in the main Agena library (lib/library.agn).

This function is system dependent and is not available on all platforms.

See also: **remove**, **select**, **os.execute**.

io.popen ([prog [, mode]])

Starts programme `prog` in a separated process and returns a file handle that you can use to read data that is sent from this programme (if `mode` is `'r'`, the default) via stdout, or to write data to this programme (if `mode` is `'w'`).

Use **io.close** to close the connection.

The following example shows how to receive the output of the UNIX \`ls\` command:

```
> p := io.popen('ls -l', 'r'):
file(779509B8)

> for keys i in io.lines(p) do print(i) od;
total 1917
drwxrwxrwx   1 user      group            0 Oct 12 17:00 OS2
-rw-rw-rw-   1 user      group        24481 Oct 13 18:23 aauxlib.c
-rw-rw-rw-   1 user      group         6205 Aug 10 02:26 aauxlib.h
-rw-rw-rw-   1 user      group        16067 Oct 12 23:42 aauxlib.o

> io.close(p):
true    0
```

This function is system dependent and is not available on all platforms.

See also: **os.execute**, **io.close**, **io.pcall**.

io.putclip (str)

Copies the string `str` to the Windows clipboard. If the clipboard could not be accessed, it returns **fail** plus an error string. It only returns fail, if something else went wrong, and **true** on success.

The function is available in the Windows edition only.

See also: **io.getclip**.

io.read (filehandle [, format])

io.read ()

In the first form, reads the file with the given `filehandle`, according to the given formats, which specify what to read. For each format, the function returns a string (or a number) with the characters read, or **null** if it cannot read data with the specified format. When called without formats, it uses a default format that reads the entire next line (see below).

The available formats are

- `'*n'`: reads a number; this is the only format that returns a number instead of a string. Hexadecimal numbers and numbers in scientific E notation are accepted, too. It also processes floats that include the decimal point separator of the current locale that may be different from a dot.
- `'*a'`: reads the whole file, starting at the current position. On end of file, it returns the empty string[22].
- `'*l'`: reads the next line (skipping the end of line), returning **null** on end of file. This is the default format.
- `'*L'`: reads the next line keeping the end-of-line character.
- **number**: reads a string up to this number of characters, returning **null** on end of file. If **number** is zero, it reads nothing and returns an empty string, or **null** on end of file.

In the second form, the function reads from the default input stream (usually the keyboard) and returns a string or number.

Note that you can write to stdin by passing the constant **io.stdin** as a file handle.

See also: **io.lines**, **io.readfile**, **io.readlines**, **skycrane.readcsv**, **utils.readcsv**, **utils.readxml**.

```
io.readfile (filename [, removenlcr [, pattern [, flag]]])
io.readfile (filhandle [, removenlcr [, pattern [, flag]]])
```

Reads the entire file with name `filename` or the file denoted by its handle `filehandle` in binary mode and returns it as a string. Note that contrary to **io.readlines**, the function also returns carriage returns (ASCII code 13).

If a second argument `removenlcr` is the Boolean value **true**, has been passed, then the function removes all newlines and if existing all carriage returns at the end of each line. If it is **false**, no such deletions are performed.

If the optional third argument `pattern` is given, the function only returns the whole contents of a file if the string `pattern` has been found in the file. Pattern matching is supported.

If the optional fourth argument `flag` is **false**, the function returns the whole file contents file if the string `pattern` has not been found in the file.

See also: **io.infile**, **io.read**, **io.readlines**, **io.writefile**.

---

[22] See also **io.readfile** to read a file entirely.

`io.readlines (filename [, options])`

`io.readlines (filehandle [, options])`

Reads the entire file with name `filename` or file handle `filehandle` and returns all lines in a table.

If a string consisting of one or more characters is given as a further argument, then all lines beginning with this string are ignored. If the option **true** is passed, then diacritics in the file are properly converted to the console character set, provided you use code page 1252. You can mix the options in any order. The function automatically deletes carriage returns (ASCII code 13) if included in the file.

You can also pass a function of one variable: in this case the function is applied on all the lines being read in and the function call results are inserted into the resulting table instead of the original lines. A line is only transformed if the line has not been skipped, see the string option mentioned above. Example:

```
> # convert all lines to upper-case, but skip all lines starting with '1':
> io.readlines(filename, '1', << x -> upper x >>):
```

An error is issued if the file could not be found.

If you use file handles, you must open the file with **io.open** before applying **io.readlines**, and close it with **io.close** thereafter.

See also: **io.lines**, **io.read**, **io.readfile**, **utils.readcsv**, **utils.readxml**, **skycrane.readcsv**.

`io.rewind (filehandle)`

Sets the current file position of the open file denoted by its `filehandle` to the beginning of the file. It returns the current file position, the number 0, at success, and **null** plus an error string otherwise.

See also: **io.move**, **io.seek**, **io.toend**.

`io.seek (filehandle [, whence [, offset]])`

Sets and gets the file position, measured from the beginning of the file, to the position given by `offset` plus a base specified by the string `whence`, as follows:

- **`'set'`:** base is position 0 (beginning of the file);
- **`'cur'`:** base is current position;
- **`'end'`:** base is end of file.

In case of success, io.seek returns the final file position, measured in bytes from the beginning of the file. If this function fails, it returns **null**, plus a string describing the error.

The default value for `whence` is `'cur'`, and for `offset` is 0. Therefore, the call `io.seek(file)` returns the current file position, without changing it; the call `io.seek(file, 'set')` sets the position to the beginning of the file (and returns 0); and the call `io.seek(file, 'end')` sets the position to the end of the file, and returns its size.

See also: **io.move**, **io.rewind**, **io.skiplines**, **io.toend**.

`io.setvbuf (filehandle, mode [, size])`

Sets the buffering mode for an output file. There are three available modes:

- **'no':** no buffering; the result of any output operation appears immediately.
- **'full':** full buffering; output operation is performed only when the buffer is full or when you explicitly flush the file (see **io.sync**).
- **'line':** line buffering; output is buffered until a newline is output or there is any input from some special files (such as a terminal device).

For the last two cases, `sizes` specifies the size of the buffer, in bytes. The default is an appropriate size.

`io.skiplines (filehandle, n)`

`io.skiplines (filename, n)`

The function skips the given number of lines and sets the file position to the beginning of the line that follows the last line skipped.

If a file name is passed, then with each call to **io.skiplines** the search always starts at the very first line in the file. The function automatically closes the file if a file name has been passed and returns the result (see below).

If you use a file handle, then lines can be skipped multiple times, always relative to the current file position. With a file handle, **io.skiplines** does not close the file.

The second argument `n` may be any non-negative number. If `n` is 0, then the function does nothing and does not change the file position.

The function returns two values: the non-negative number of lines actually skipped and the non-negative number of characters skipped in this process, including newlines and carriage returns.

See also: **io.nlines**, **io.seek**.

`io.sync (filehandle)`

`io.sync ()`

In the first form, saves any written data to the file denoted by `filehandle`. In the second form, the function flushes the default output.

`io.tmpfile ()`

Returns a handle to a temporary file. The file is opened in update mode and it is automatically removed when the programme ends.

See also: **io.mkstemp**, **os.tmpdir**, **os.tmpname**.

`io.toend (filehandle)`

Sets the current file position of the open file denoted by its `filehandle` to the end of the file. It returns the current file position, a number indicating the size of the file, at success, and **null** plus an error string otherwise.

See also: **io.move**, **io.rewind**, **io.seek**.

`io.unlock (filehandle [, size])`

The function unlocks the file given by its handle `filehandle` so that it can be read or overwritten by other applications again. If size is given, the function, only the given number of bytes is unlocked, starting from the current file position.

The function returns **true** on a successful unlock, and **false** otherwise.

For more information, see **io.lock**.

`io.write (···)`

`io.writeline (···)`

Write the value of each of its arguments to standard output if the first argument is not a file handle, or to the file denoted by the first argument, a file handle. Except for the file handle and the `'delim'` option described below, all arguments must be strings, numbers, Booleans or **null**. To write other values, use **tostring** or **strings.format**.

io.writeline adds a new line at the end of the data written, whereas **io.write** does not.

By default, no character is inserted between neighbouring values. This may be changed by passing the option `'delim':<str>` (i.e. a pair, e.g. `'delim':'|'`) as the last argument to the functions with `<str>` being a string of any length. Remember that in the function call, a shortcut to `'delim':<str>` is `delim ~ <str>`.

The functions return **true** on success, and **false** otherwise.

Hint: If you work in DOS-like systems, such like DOS, Windows, or OS/2, and if the text to be written includes line breaks, you may wonder why the resulting file will be larger than the number of characters in the text. This is because the operating system adds a further control code, i.e. carriage return, in front of each line break. To avoid this, open the file in binary mode, e.g. `io.open(filename, 'wb')`.

Examples:

Write a string to the console. Note that in the first statement, no newline is added to the output, as opposed to the second and third statements.

```
> io.write('Gauden Dach !')
Gauden Dach !

> io.write('Gauden Dach !', '\n')
Gauden Dach !

> io.writeline('Gauden Dach !')
Gauden Dach !
```

Write strings to the console:

```
> io.writeline('Bet', 'to\'n', '16.', 'Johrhunnert', 'geef', 'dat', 'hier',
>    'baben', 'anne', 'Küst', 'nix', 'anneres', 'as', 'Platt.')
Betto'n16.JohrhunnertgeefdathierbabenanneKüstnixanneresasPlatt.
```

Use a white space as a separator:

```
> io.writeline('Bet', 'to\'n', '16.', 'Johrhunnert', 'geef', 'dat', 'hier',
>    'baben', 'anne', 'Küst', 'nix', 'anneres', 'as', 'Platt.',
>    delim=' ')
Bet to'n 16. Johrhunnert geef dat hier baben anne Küst nix anneres as
Platt.
```

Write a string to a new file called `'d:/newfile.txt'`: First we have to create the new file with **io.open** and the `'w'` (write) option.

```
> fh := io.open('d:/newfile.txt', 'w'):
file(7803A6F0)
```

Write some text to the file.

```
> io.write(fh, 'Gouden Dach !'):
true

> io.writeline(fh, '\nBet', 'to\'n', '16.', 'Johrhunnert', 'geef', 'dat',
>    'hier', 'baben', 'anne', 'Küst', 'nix', 'anneres', 'as', 'Platt.',
>    delim=' '):
true
```

Finally, the file will be closed.

```
> io.close(fh):
true
```

Note that you can also write to stdin, stdout and stderr by passing the constants **io.stdin**, **io.stdout** or **io.stderr** as a file handle.

See also: **io.writefile**, **print**, **skycrane.formatline**, **skycrane.scribe**, **skycrane.tee**.

io.writefile (filename, ···)
io.writefile (filehandle, ···)

In the first form, creates a new file `filename` denoted by its first argument (a string) and writes all of the given strings or numbers starting with the second argument in binary mode to it. To write other values, use **tostring** or **strings.format**. After writing all data, the function automatically closes the new file.

In the second form, the function writes its arguments to the open file denoted by its handle `filehandle`.

By default, no character is inserted between neighbouring strings. This may be changed by passing the option 'delim':<str> (i.e. a pair, e.g. 'delim':'|') as the last argument to the function with <str> being a string of any length.
If the file `fn` already exists, it is overwritten without warning.

The function returns the total number of bytes written, and issues an error otherwise. It is around twice as fast than using a combination of **io.open**, **io.write**, and **io.close**.

See also: **save**, **io.readfile**.

12.2 binio - Binary File Package

This package contains functions to read data from and write data to binary files.

Summary of functions:

Opening and closing files:

> **binio.open**, **binio.close**, **binio.isfdesc**.

Reading data:

> **binio.lines**, **binio.readbytes**, **binio.readchar**, **binio.readlong**, **binio.read-longdouble**, **binio.readnumber**, **binio.readshortstring**, **binio.readstring**.

Writing data:

> **binio.writebytes**, **binio.writechar**, **binio.writeline**, **binio.writelong**, **binio.write-longdouble**, **binio.writenumber**, **binio.writeshortstring**, **binio.writestring**.

File positions:

> **binio.eof**, **binio.filepos**, **binio.rewind**, **binio.seek**, **binio.toend**.

File locking:

> **binio.lock**, **binio.unlock**.

File buffering:

> **binio.sync**.

Miscellaneous:

> **binio.length**.

The binio package always uses file handles that are positive integers greater than 2. (Note that the **io** package uses file handles of type userdata.) The positive integer will be returned by the **binio.open** function and must be used in all package functions that require a file handle.

A typical example might look like this:

Open a file and return the file handle:

```
> fh := binio.open('c:/agena/lib/library.agn'):
3
```

Determine the size of the file in bytes:

```
> binio.length(fh):
46486
```

Close the file.

```
> binio.close(fh):
true
```

binio supports metamethods. The metatable used by the package is called `BINIOFILE*`. By default, only __gc and __tostring methods are supported. Check the end of Chapter 6.19 on how to add further methods.

The **binio** functions are:

binio.close (filehandle [, filehandle2, ···])

Closes the files identified by the given file handle(s) and returns **true** if successful, and issues an error otherwise. The function also deletes the file handles and the corresponding filenames from the **binio.openfiles** table if the file could be properly closed.

See also: **binio.open**.

binio.eof (filehandle)

Checks whether the end of the file denoted by filehandle has been reached and returns **true** or **false**.

binio.filepos (filehandle)

Returns the current file position relative to the beginning of the file as a number. In case of an error, it quits with this error.

binio.isfdesc (filehandle)

Checks whether filehandle is a valid file handle. Returns **true** if filehandle is an open file handle, or **false** if filehandle is not a file handle.

binio.length (filehandle)

The function returns the size of the file denoted by filehandle in bytes. In case of an error, it quits with this error.

```
binio.lines (filehandle [, n] [, true])
```

Creates an iterator function that beginning from the current file position, with each call will return a new line from the file pointed to by the handle `filehandle`.

By default, the function traverses the file up to its end. If the second argument `n` is a positive integer, it will read the next `n` characters from the current file position (default is **infinity** = end of file). The function generally ignores carriage returns (ASCII code 13) and does not return newlines (ASCII code 10).

If the last argument is the Boolean value **true**, all embedded zeros (ASCII Code 0) will be replaced with white spaces, and the traversal of the file will continue. By default, zeros are not ignored, so if one is found, the traversal will stop.

The iterator function returns a string, and **null** if the end of the file has been reached. It also returns **null** if the last argument is not **true** and an embedded zero has been found in the file.

The iterator function does not close the file at the end of traversal, use **binio.close** to accomplish this.

```
binio.lock (filehandle)
```

```
binio.lock (filehandle, size)
```

The function locks the file given by its handle `filehandle` so that it cannot be read or overwritten by other applications.

In the first form, the entire file is locked in UNIX-based systems. In Windows, only 2^{63} bytes are locked, so you have to use the second form in Windows after the file has become larger than 2^{63} bytes (= 8,589,934,592 Gbytes).

In the second form the function locks `size` bytes from the current file position. Locked blocks in a file may not overlap. `size` may be larger than the current file length.

The function returns **true** on a successful lock, and **false** otherwise.

Note that other applications that do not use the locking protocol may nevertheless have read and write access to the file.

See also: **binio.unlock**.

```
binio.open (filename, 'a')
binio.open (filename [, anything else])
```

Opens the given file denoted by `filename` and returns a file handle (a number).

If it cannot find the file, it creates it and leaves it open for further binio operations.

In the first form, by passing the `'a'` or `'append'` option, and if the file already exists, it opens the file and sets the current file position to the end of the file so that nothing will be overwritten later on.

In the second form, if the file already exists, it opens the file and sets the current file position to the beginning of the file. In subsequent write operations, the contents of the file will thus be overwritten and the programmer has to ensure its integrity himself. (Use **binio.toend** to append to the file in this case or pass the `'a'` option.)

The file is always opened in both read and write modes.

If an optional second argument except `'a'` or `'append'` is given, the file is opened in read mode only. Thus, if the file does not yet exist, the function returns an error.

The function also enters the newly opened file into the **binio.openfiles** table.

See also: **binio.close**, **binio.lock**, **binio.unlock**, **os.exists**.


```
binio.readbytes (filehandle [, bytes] [, options])
```

By default, the function reads **environ.kernel('buffersize')** bytes from the file denoted by `filehandle` and returns them as a sequence of integers. You may change the kernel buffer size value to any other values in order to read less or more bytes.

If `bytes` is given, the function reads `bytes` bytes from the file denoted by `filehandle` and returns them as a sequence of integers.

The function increments the file position thereafter so that the next bytes in the file can be read with a new call to various **binio.read*** functions.

If the end of the file has been reached, **null** will be returned. In case of an error, it quits with the respective error.

By default, the function reads in all bytes of a file, including newlines (ASCII 10) or carriage returns (ASCII 13). You can change this by setting the `ignore` option and passing a string of explicit bytes that shall be skipped, e.g.:

```
> binio.readbytes(fh, ignore=" .\n");  # skip white space, dot & newline
```

Also by default, the function reads in embedded zeros and treats them as every other byte. If you pass the `eof` option and set it to **true**, then the function quits if it

encounters an embedded zero in the file. The file pointer is automatically reset to the position of the embedded zero. The default is **false**, i.e. the whole file is read in.

The function is much faster when working on a larger number of bytes.

See also: **binio.writebytes**, **bytes.tonumber**, **stack.readbytes**, **strings.tochars**.

`binio.readchar (filehandle)`

`binio.readchar (filehandle, offset)`

In the first form, the function reads a byte from the file denoted by `filehandle` from the current file position and increments the file position thereafter so that the next byte in the file can be read with a new call to **binio.read*** functions.

In the second form, at first the file position is changed by `offset` bytes (a positive or negative number or zero) relative to the current file position. After that, the byte at the new file position is read. Next, the file position is being incremented thereafter so that the next byte in the file can be read with a new function call.

If the byte is successfully read, it will be returned as a number. If the end of the file has been reached, **null** will be returned. In case of an error, the function quits.

`binio.readindex (filehandle , k [, type [, offset]])`

The function assumes that all values in the binary file pointed to be `filehandle` are of the same type and reads the `k`-th one. By default, the function reads numbers (C doubles). You may pass the third argument `type` to determine another type. Valid types are the strings 'char' (see **binio.writechar**), 'long' (see **binio.writelong**), 'number' (the default, see **binio.writenumber**), 'shortstring' (see **binio.writeshortstring**) or 'string' (see **binio.writestring**). Longdoubles are not supported.

You may pass an optional offset from the beginning of the file as the fourth argument, which by default is 0. If given, the file position is moved the offset's + 1 byte in the file before searching for the given index and reading the value of interest. This feature supports a self-defined file header.

See also: **binio.readbytes**, **binio.readchar**, **binio.readlong**, **binio.readnumber**, **binio.readshortstring**, **binio.readstring**, **binio.writeindex**.

`binio.readlong (filehandle [, offset])`

The function reads a signed C value of type `int32_t` from the file denoted by `filehandle` from the current file position and returns it. If there is nothing to read, the function returns **null**. Note that the number to be read should have been written to the file using the **binio.writelong** function.

In the second form, before reading the actual value, at first the file position is changed by `offset` bytes (a positive or negative number or zero) relative to the current file position.

See also: **binio.writelong**.

`binio.readlongdouble (filehandle [, offset])`

The function reads a longdouble from the file denoted by `filehandle` from the current file position and returns it. If there is nothing to read, the function returns **null**. Note that the longdouble to be read should have been written to the file using the **binio.writelongnumber** function. See the **long** package for further information on 80-bit floating point values.

In the second form, before reading the actual value, at first the file position is changed by `offset` bytes (a positive or negative number or zero) relative to the current file position.

The function is not supported on Big Endian systems.

See also: **binio.writelongdouble**.

`binio.readnumber (filehandle [, offset])`

The function reads an Agena number from the file denoted by `filehandle` from the current file position and returns it. If there is nothing to read, the function returns **null**. Note that the number to be read should have been written to the file using the **binio.writenumber** function.

In the second form, before reading the actual value, at first the file position is changed by `offset` bytes (a positive or negative number or zero) relative to the current file position.

See also: **binio.writenumber**.

`binio.readshortstring (filehandle)`

The function reads a string of up to 255 characters from the file denoted by `filehandle` from the current file position and returns it. If there is nothing to read, the function returns **null**.

Note that the string to be read should have been written to the file using the **binio.writeshortstring** function, as this function also stores the length of the string in a special way to the file.

See also: **binio.writeshortstring**.

`binio.readstring (filehandle)`

The function reads a string of any length from the file denoted by `filehandle` from the current file position and returns it. If there is nothing to read, the function returns **null**.

Note that the string to be read should have been written to the file using the **binio.writestring** function, as this function also stores the length of the string in a special way to the file.

See also: **binio.writestring**.

`binio.rewind (filehandle [, pos])`

Sets the file position to the beginning of the file denoted by `filehandle`.

If `pos`, a non-negative integer is given, the function resets the file pointer to the position `pos` relative to the beginning of the file.

The function returns the new file position as a number in case of success, and quits with an error otherwise.

See also: **binio.toend**, **binio.seek**.

`binio.seek (filehandle, position)`

The function changes the file position of the file denoted by `filehandle position` bytes relative to the current position. `position` may be negative, zero, or positive.

The return is **true** if the file position could be changed successfully, or issues an error otherwise.

See also: **binio.rewind**, **binio.toend**.

`binio.sync (filehandle)`

Flushes all unwritten content to the file denoted by the handle `filehandle`. The function returns **true** if successful, **false** if stdin or stdout should be closed, and issues an error otherwise (e.g. if the file was not opened before or an error during flushing occurred).

`binio.toend (filehandle)`

Sets the file position to the end of the file denoted by `filehandle` so that data can be appended to the file without overwriting existing data. The function returns the file position as a number in case of success, and issues an error otherwise.

See also: **binio.rewind**, **binio.seek**.

`binio.unlock (filehandle)`

`binio.unlock (filehandle, size)`

The function unlocks the file given by its handle `filehandle` so that it can be read or overwritten by other applications again.

The function returns **true** on a successful unlock, and **false** otherwise.

For more information, see **binio.lock**.

`binio.writebytes (filehandle, s)`

The function writes all integers in the sequence `s` to the file denoted by `filehandle` at its current position. The function returns **true** in case of success and **fail** if the sequence is empty.

The integers in `s` should be integers `number` with $0 \leq$ `number` < 256, otherwise `number` % 256 will be stored to the file.

Internally, the bytes are stored as C `unsigned char`'s.

See also: **binio.readbytes**, **math.tobytes**, **strings.tobytes**.

`binio.writechar (filehandle, number [, ···])`

The function writes the given Agena `number`, and optionally more numbers, to the file denoted by `filehandle` at its current position. The function returns **true** in case of success and **quits with an error** otherwise.

All `number`(s) should be integers with $0 \leq$ `number` < 256, otherwise `number` % 256 will be stored to the file.

Internally, the bytes are stored as a C `unsigned char`.

`binio.writeindex (filehandle, k, type, value [, offset])`

The function assumes that all values in the binary file pointed to by `filehandle` are of the same type and writes the `k`-th one.

The third argument `type` specifies the type to be written. Valid types are the strings 'char' (see **binio.writechar**), 'long' (see **binio.writelong**), 'number' (see **binio.writenumber**), 'shortstring' (see **binio.writeshortstring**) or 'string' (see **binio.writestring**). Longdoubles are not supported.

The fourth argument specifies the actual `value` to be written.

You may pass an optional `offset` from the beginning of the file as the fifth argument, which by default is 0. If given, the file position is moved to the `offset`'s + 1 byte before writing a value. This feature allows for a self-defined file header.

See also: **binio.writebytes**, **binio.writechar**, **binio.writelong**, **binio.writenumber**, **binio.writeshortstring**, **binio.writestring**, **binio.readindex**.

binio.writeline (filehandle, ···)

Writes one or more strings to the file denoted by its file handle `filehandle`, separated by newlines.

The function is written in the Agena language and is included in the lib/library.agn file.

binio.writelong (filehandle, number [, ···])

The function writes the given Agena `number`, and optionally more numbers, to the file denoted by `filehandle` at its current position. The `number`(s) should be integers with **environ.minlong** < `number` < **environ.maxlong**, otherwise the result is not defined.

The function returns **true** in case of success and quits with an error otherwise.

Internally, the numbers are stored as signed C `int32_t` in Big Endian notation. Use **binio.readlong** to read values written by **writelong** back into Agena as **readlong** transforms the value back into the proper Endian format used by your machine.

binio.writelongdouble (filehandle, number [, ···])

The function writes the given longdouble `number`, and optionally more longdoubles, to the file denoted by `filehandle` at its current position. The function returns **true** in case of success and issues an error otherwise. The function is not supported on Big Endian systems. See also: **binio.readlongdouble**.

binio.writenumber (filehandle, number [, ···])

The function writes the given Agena `number`, and optionally more numbers, to the file denoted by `filehandle` at its current position. The function returns **true** in case of success and issues an error otherwise. The numbers are always stored in Big Endian notation. The **binio.readnumber** function conducts proper conversion to Little Endian if Agena runs on a Little Endian machine.

`binio.writeshortstring (filehandle, string [, ···])`

The function writes the given `string`, and optionally more strings, to the file denoted by `filehandle` at its current position. The strings can be of length 0 to 255.

The function returns **true** in case of success and issues an error otherwise. Internally, **writeshortstring** at first writes the length of the respective string as a C unsigned char and after this it stores the string without a trailing null character to the file. If you call **binio.readstring** later, Agena very efficiently returns the string.

See also: **binio.readshortstring**.

`binio.writestring (filehandle, string [, ···])`

The function writes the given `string`, and optionally more strings, to the file denoted by `filehandle` at its current position.

The function returns **true** in case of success and quits with an error otherwise. Internally, **writestring** first writes the length of the respective string as a C long int and then the string without a null character to the file. This information is then read by the **binio.readstring** function to efficiently return the string.

See also: **binio.readstring**.

12.3 xbase - Library to Read and Write xBase Files

As a *plus* package, in Solaris, Linux, Mac OS X, and Windows, this library is not part of the standard distribution and must be activated with the **import** statement, e.g. `import xbase`.

This package provides basic functions to read and write dBASE III+ compliant files.

A typical session may look like this:

```
> import xbase alias;

> new('test.dbf', data=Number);

> f := open('test.dbf', 'write');

> writenumber(f, 1, 1, Pi);

> readvalue(f, 1, 1):
3.1415926535898

> close(f):
true
```

Limitations:

1. The xBase data types currently supported are: Number, Float (dBASE IV 2.0), Binary Double (dBASE 7), String, Date, and Logical.
2. Only files with extension .dbf are supported. Searching and sorting functions are not available, and any .ndx, or .idx index files or *.dbt files will be ignored.
3. Files with sizes greater than 2 GBytes are not supported.

xbase.attrib (filehandle)

returns a table with various information on the xBase file pointed to by `filehandle`.

Table key	Meaning
`'codepage'`	Code page used.
`'fieldinfo'`	A table of tables that describe the respective fields in consecutive order: title, xBase native type (see below), Agena type, total number of bytes occupied by the field in the file. With numbers, the number of decimals following the decimal point (its scope) given.
`'fields'`	Number of fields in the file.
`'filename'`	Name of the xBase file (relative).
`'headerlength'`	Length of the header in the xBase file.
`'lastmodified'`	UTC date of the last write access, coded as an integer.
`'records'`	Number of records stored in the file.
`'recordlength'`	Number of bytes occupied by each record.
`'version'`	dBASE version number (see **xbase.new**)
`'versionname'`	dBASE version name (a string, see **xbase.new**)

xBase native types recognised are: 'C' for String, 'N' for Number, 'F' for Float, 'L' for Logical, 'D' for Date, and 'O' for a binary Double and 'I' for a binary 4-byte signed integer. 'B' indicates a .DBT block number and 'M' a memo field. See **xbase.new** for further information.

For known version numbers, see **xbase.new**, as well. To check for dBASE 7, binary-and the result with 0b111 and check for result 4, e.g. result **&&** 0b111 = 4.

See also: **xbase.fieldtype**, **xbase.filepos**.

xbase.close (filehandle)

Closes a connection to the xBase file pointed to by `filehandle`. No more data can be read or written to the xBase file until you open it again using **xbase.open**. The function returns **true** if the file could be closed, and **false** otherwise.

xbase.field (filehandle, row [, 'set'])

The function has been deprecated. Please use **xbase.readdbf** instead.

See also: **xbase.ismarked**, **xbase.readdbf**, **xbase.readvalue**, **xbase.record**.

xbase.fields (filehandle)

Returns the number of fields per record contained in the xBase file denoted by `filehandle`.

See also: **xbase.attrib**, **xbase.records**.

xbase.fieldtype (filehandle, field)

Determines the dBASE data type of the given `field` in the open file denoted by `filehandle`. The function returns a one-character string, or the string `'?'` if it is unknown. See **xbase.new** for the meaning of the return.

See also: **xbase.attrib**.

xbase.filepos (filehandle)

Returns the current file position in the file denoted by `filehandle` and returns it as a number.

See also: **xbase.attrib**.

`xbase.header (filehandle)`

Returns three sequences: the header field names of the file denoted by `filehandle`, the corresponding Agena type names, and the respective single-character dBASE types.

See also: **xbase.attrib**.

`xbase.ismarked (filehandle, record)`

Checks whether a record in a file denoted by `filehandle` has been marked as to be deleted and returns **true** or **false**.

Please make sure that the file has been opened in write, append, or read/write mode before, otherwise the result may be undefined.

See also: **xbase.mark**.

`xbase.isopen (filehandle)`

Checks whether `filehandle` points to an open xBase file (opened by the same Agena session) and returns **true** or **false**.

`xbase.isvoid (filehandle, record, field)`

Checks whether the value at record number `record` and field number `field` from the file pointed to by `filehandle` has been deleted.

The function returns either **true** or **false**.

See also: **xbase.ismarked**, **xbase.mark**, **xbase.purge**, **xbase.readvalue**.

`xbase.kernel ([options])`

The function sets defaults for the binary representation and layout of binary doubles and binary time stamps in dBASE Level 7 files:

Setting	Default	Description
`DoubleIsBigEndian`	**true**	If set to **false**, binary doubles shall be represented in Little Endian notation. By default, binary doubles are Big Endian.
`LongIsBigEndian`	**false**	If set to **true**, binary doubles shall be represented in Big Endian notation. By default, binary integers are Little Endian.
`TimestampIsDouble`	**false**	If set to **true**, binary time stamps are represented as a binary 8-byte double; by default time and date are represented by two 4-byte signed integers, with the endianness depending on the setting of `Long1IsBigEndian`.

If no argument is given, the current settings are returned.

`xbase.lock (filehandle)`

`xbase.lock (filehandle, size)`

The function locks the file given by its handle `filehandle` so that it cannot be read or overwritten by other applications.

In the first form, the entire file is locked in UNIX-based systems. In Windows, only 2^{63} bytes are locked, so you have to use the second form in Windows after the file has become larger than 2^{63} bytes (= 8,589,934,592 GBytes).

In the second form the function locks `size` bytes from the current file position. Locked blocks in a file may not overlap. `size` may be larger than the current file length.

The function returns **true** on success and **false** otherwise.

Note that other applications that do not use the locking protocol may nevertheless have read and write access to the file.

See also: **xbase.unlock**.

`xbase.mark (filehandle, row [, flag])`

Marks the record number `row`, an integer, in the file denoted by its `filehandle`, as deleted.

Returns **true** if a record has been marked successfully, **and false** otherwise.

The actual data is not physically deleted, however, **xbase.readvalue**, **xbase.record**, **xbase.field**, and **xbase.readdbf** do not return it. Use **xbase.purge** to delete entries.

If flag is **false**, a formerly marked record is activated (`undeleted`) again.

Please make sure that the file has been opened in write, append, or read/write mode before, otherwise the result may be undefined.

See also: **xbase.ismarked**.

`xbase.new (filename, desc₁ [, code page] [, version] [, desc₂, ···, descₖ])`

creates a new xBase file with the file name `filename`.

`descₖ` are `k` fields (columns) the xBase file will contain. code page indicates the code page to be used (see below)[23].

In its header, the function designates the resulting file as a dBASE III+ file without memo .DBT file.

`descₖ` must be a pair of the following form:

1. field_name : data_type

    where `field_name` is a string and the name of the field to be added, and `data_type` is one of the strings `'Logical'`, `'Date'`, `'Float'`, `'Number'`, `'Double'`, `'Long'`, `'Complex'`, `'Byte'`, `'Decimal'` or `'Character'`, i.e. the xBase data type of the values to be stored later.

    Examples:

    ```
    new('dbase.dbf', 'logical':'Logical') or
    new('dbase.dbf', logical='Logical') for short for a Boolean.
    ```

    A Boolean (which in xBase is equal to a `Logical`) will always consist of one character `'T'`, `'F'` for **true** and **false**.

    An xBase Number will have a standard length of 19 places with a default scale of 15 digits, whereas an xBase Float consists of 20 places with a scale of 18 digits (scale: numbers following the decimal point). Numbers are stored in xBase files as strings with ANSI C double precision. The scale may be in [0, 15] with xBase Numbers, and in [0, 18] with xBase Floats.

    An xBase Character (string) will have a default length of 64 characters. The minimum length of a string is 1, the maximum length of a string may be 254 characters. Longer strings will be truncated.

    A date will always consist of eight digits of the format YYYYMMDD.

    A dBase Level 7 Double represents an Agena number (integral or fractional) that is stored in either binary Big Endian or Little Endian format of eight bytes to an xBase file. The default is Big Endian, but you can change this when writing or reading files, see **xbase.kernel**.

    A dBase Level 7 Long represents a signed 4-byte integer that is stored in either binary Big Endian or Little Endian format to an xBase file. The default is Little Endian, but you can change this when writing or reading files, see **xbase.kernel**.

---

[23] Note that code pages are a Foxpro extension.

There are three proprietary nonstandard types: Type `'Complex'` represents a complex number, stored in 16 bytes, type `'Byte'` represents an unsigned integer in the range 0 .. 255, stored in just one byte, and the experimental type `'Decimal'` stores numbers as signed 4-byte C floats (note that these are highly inaccurate). All of these three types are represented in Little Endian notation.

1. field_name : data_type : length

    where `field_name` and `data_type` are the same as mentioned above, and `length` is the maximum length of the item to be added. `length` must be a positive integer. With numbers, `length` denotes the number of digits after the decimal point to be stored.
    When passing a length value, you may leave out the quotes for data_type values.

    Examples:

    ```
    new('dbase.dbf', 'value':'Number':5) or
    new('dbase.dbf', value=Number:5) for short for a float with five decimal places.
    ```

Supported xBase data types are:

Type	data_type name	Agena type	Write function	dBASE version
Logical	`'Logical'` or `'L'`	boolean	**xbase.writeboolean**	III+
Number	`'Number'` or `'Numeric'` or `'N'`	number	**xbase.writenumber**	III+
Float	`'Float'` or `'F'`	number	**xbase.writefloat**	IV 2.0
Double	`'Double'` or `'O'`	number	**xbase.writedouble**	7
Long	`'Long'` or `'I'`	number	**xbase.writelong**	7
Character	`'Character'` or `'C'`	string	**xbase.writestring**	III+
Date	`'Date'` or `'D'`	string	**xbase.writedate**	III+
Timestamp	`'@'`	numbers	**xbase.writetime**	7
OLE	`'G'`		n/a	?
Memo	`'M'`		n/a	?
Binary	`'B'`		n/a	?
Complex	`'Complex'` or `'c'`	complex	**xbase.writecomplex**	prop.
Byte	`'Byte'` or `'b'`	number	**xbase.writebyte**	prop.
Decimal	`'Decimal'` or `'f'`	number	**xbase.writedecimal**	prop.

`codepage` should be a pair of the form `'codepage'`:n, with n an integer in [0, 255].

Valid codepages are:

n	Meaning	Code page
0x01	DOS USA	437
0x02	DOS Multilingual	850
0x03	Windows ANSI	1.252
0x04	Standard Macintosh	10.000
0x64	Eastern Europe DOS	852
0x65	Nordic DOS	865
0x66	Russian DOS	866
0x67	Icelandic DOS	861
0x68	Kamenicky (Czech) DOS	895
0x69	Mazovia (Polish) DOS	620
0x6a	Greek DOS	437G
0x6b	Turkish DOS	857
0x78	Traditional Chinese (Republic of China (Taiwan), Hong Kong SAR)	950
0x79	Korean Windows	949
0x7A	Chinese Simplified (Singapore, PRC)	936
0x7B	Japanese Windows	932
0x7C	Thai Windows	874
0x7D	Hebrew Windows	1.255
0x7E	Arabic Windows	1.256
0x96	Russian Macintosh	10.007
0x97	Eastern European Macintosh	10.029
0x98	Greek Macintosh	10.006
0xc8	Eastern Europe Windows	1.250
0xc9	Russian Windows	1.251
0xca	Turkish Windows	1.254
0xcb	Greek Windows	1.253

If no code page has been passed, it is set to 0x00.

Example for Eastern European Macintosh:

```
new('dbase.dbf', text=string:255, code page=0x97);
```

`version` should be a pair of the form `'version'`:n, with n an integer in [0, 255].

dBASE version numbers are:

```
x xxx x 001 = 0x?1 not used
0 000 0 010 = 0x02 FoxBASE
0 000 0 011 = 0x03 FoxBASE+/dBASE III PLUS, no memo
x xxx x 100 = 0x?4 dBASE 7
0 000 0 101 = 0x05 dBASE 5, no memo
0 011 0 000 = 0x30 Visual FoxPro
0 011 0 001 = 0x31 Visual FoxPro, autoincrement enabled
0 011 0 010 = 0x32 Visual FoxPro, Varchar, Varbinary, or Blob-enabled
0 100 0 011 = 0x43 dBASE IV SQL table files, no memo
0 110 0 011 = 0x63 dBASE IV SQL system files, no memo
0 111 1 011 = 0x7B dBASE IV, with memo
1 000 0 011 = 0x83 FoxBASE+/dBASE III PLUS, with memo
1 000 1 011 = 0x8B dBASE IV, with memo
1 000 1 110 = 0x8E dBASE IV with SQL table
1 100 1 011 = 0xCB dBASE IV SQL table files, with memo
1 110 0 101 = 0xE5 Clipper SIX driver, with SMT memo
1 111 0 101 = 0xF5 FoxPro 2.x (or earlier) with memo
1 111 1 011 = 0xFB FoxBASE (with memo?)
| ||| | |||
| ||| | |||   Bit flags (not used in all formats)
| ||| | |||   --------------------------------
| ||| | ++--- bits 2, 1, 0, version (x03 = level 5, x04 = level 7)
| ||| +------ bit 3, presence of memo file
| +++-------- bits 6, 5, 4, presence of dBASE IV SQL table
+----------- bit 7, presence of .DBT file
```

The default is 0x03 = 3 decimal for dBASE III+. If at least one of the given fields is of dBASE data type 'Double' (= 'B' or 'O') or type 'timestamp', then the version number is automatically changed to 0x?4 = 4 decimal = dBASE Level 7. This allows dBASE files created with Agena and containing binary Doubles to be imported into LibreOffice 5.x. Current versions of Excel still cannot read Visual Fox Pro dbf files with Doubles or Longs, so you might pass the `version` option.

See also: **xbase.open**.

xbase.open (filename [, mode])

Opens an xBase file of the name `filename` for reading or writing, or both.

In the first form, the file is opened for reading only.

In the second form, if `mode` is either `'write'`, `'w'`, `'append'`, or `'r+'`, the file is opened for reading while new data sets may be added to the end of the file.

If mode is `'read'` or `'r'`, the file is opened for reading only.

The return is a file handle to be used by all other xBase package functions.

See also: **xbase.close**, **xbase.lock**, **xbase.new**.

`xbase.purge (filehandle, record, field)`

Overwrites the specified `field` in the given `record` of the file denoted by its handle `filehandle` with asterisks, thus physically deleting the original content. The return is **true** if deletion succeeded, and **false** otherwise. After successful completion, a subsequent call to **xbase.isvoid** would return **true**.

See also: **xbase.isvoid**, **xbase.mark**, **xbase.wipe**.

`xbase.readdbf (filename [, option])`
`xbase.readdbf (filehandle [, option])`

In the first form, opens an xBase file denoted by its `filename` in read mode, returns all its records and fields, and closes it. In the second form, it reads the contents of the open file denoted by its handle `filehandle`.

If the xBase file contains more than one field, the data will be returned as a sequence of sequences, whereas if the file contains only one field, all values are returned in one sequence only.

If the option fields=x with x a positive number is given, only the given column x is extracted, and the return is a sequence of the column values. If the option fields=obj with obj a table or sequence of positive numbers is given, only the given fields in the records are returned, and the return is a sequence of sequences.

If a record has been marked as being deleted, the function ignores the record.

See also: **xbase.field**, **xbase.ismarked**, **xbase.readvalue**, **xbase.record**.

`xbase.readvalue (filehandle, record, field)`

Reads a value at record number `record` and field number `field` from the file pointed to by `filehandle`.

Supported values are of xBase type Logical, Number, Float, binary double, Date, Timestamp and String. Also Binary, Memo and OLE .DBT block numbers are supported, as well as type Complex, Byte and Decimal. If a number could not be read from the file, the function returns 0. On Little Endian systems, you might have to convert Big Endian binary doubles back to Little Endian by calling **bytes.tolittle**; see also **os.endian**.

If `record` has been marked as being deleted, the function returns **null**.

See also: **xbase.field**, **xbase.ismarked**, **xbase.record**, **xbase.isvoid**.

`xbase.record (filehandle, line)`

Returns all values in the given record `line` (a number) of the file denoted by `filehandle` and returns them in a sequence.

If `record` has been marked as being deleted, the function returns **null**.

See also: **xbase.field**, **xbase.ismarked**, **xbase.readdbf**, **xbase.readvalue**.

`xbase.records (filehandle)`

Returns the number of records contained in the xBase file denoted by `filehandle`, including the ones marked as to be deleted or being completely void.

See also: **xbase.attrib**, **xbase.fields**.

`xbase.sync (filehandle)`

Writes any unwritten content to the xBase file pointed to by `filehandle`. The function either returns **true** if flushing succeeded or nothing had be flushed, or **fail** otherwise.

Please make sure that the file has been opened in write, append, or read/write mode before, otherwise the result may be undefined.

`xbase.unlock (filehandle)`

`xbase.unlock (filehandle, size)`

The function unlocks the file given by its handle `filehandle` so that it can be read or overwritten by other applications again.

The function returns **true** on success and **false** otherwise.

For more information, see **xbase.lock**.

`xbase.wipe (filehandle, record)`

In an xBase file denoted by `filehandle`, deletes all fields of the given `record`, a positive integer by overwriting all fields with asterisks. It also marks the record as deleted (see **xbase.mark** for further information).

To ensure performance, the function does not lock the file before deleting data - you may want to manually call **xbase.lock** before and xbase.unlock thereafter. Also, it does not flush the file.

The function returns nothing.

The function has been written in Agena, see lib/xbase.agn.

See also: **xbase.mark**, **xbase.purge**.

xbase.writeboolean (filehandle, record, field, value)

Writes the Boolean value **true** or **false** (4th argument) to the file denoted by `filehandle` to record number `record` and field number `field`. **fail** and **null** are not supported.

When creating the dBASE file with **xbase.new**, pass the `'L'` data type descriptor for the respective fields.

The return is **true** if writing succeeded, and **false** otherwise.

xbase.writebyte (filehandle, record, field, value)

Writes the integer `value` in the range 0 .. 255(4th argument) to the file denoted by `filehandle` to record number `record` and field number `field`.

When creating the dBASE file with **xbase.new**, pass the `'b'` data type descriptor for the respective fields.

The integer is stored in binary format of just one byte (unsigned C char).

A proprietary extension, applications that import dBASE files - such as Microsoft Excel - do not support this type.

The return is **true** if writing succeeded, and **false** otherwise. Note that the return **false** only indicates that an error may have occurred.

See also: **xbase.writefloat**, **xbase.writelong**, **xbase.writenumber**, **xbase.new**.

xbase.writecomplex (filehandle, record, field, value)

Writes the complex number `value` (4th argument) to the file denoted by `filehandle` to record number `record` and field number `field`.

When creating the dBASE file with **xbase.new**, pass the `'c'` data type descriptor for the respective fields.

The complex number is stored in Little Endian binary format of sixteen bytes (two C doubles, converted to two C uint64_t).

A proprietary extension, applications that import dBASE files - such as Microsoft Excel - do not support this type.

The return is **true** if writing succeeded, and **false** otherwise. Note that the return **false** only indicates that an error may have occurred.

See also: **xbase.writefloat**, **xbase.writelong**, **xbase.writenumber**, **xbase.new**.

`xbase.writedate (filehandle, record, field, value)`

Writes the string or number `value` (4th argument), representing an integer - or a string representing an integer - in the range $19000101 \leqslant x \leqslant 99991231$ and denoting a date, to the file denoted by `filehandle` to record number `record` and field number `field`.

When creating the dBASE file with **xbase.new**, pass the `'D'` data type descriptor for the respective fields.

The return is **true** if writing succeeded, and **false** otherwise. Note that the return **false** only indicates that an error may have occurred.

See also: **xbase.writetime**.

`xbase.writedecimal (filehandle, record, field, value)`

Writes the number `value` (4th argument) as a signed 4-byte Little Endian float to the file denoted by `filehandle` to record number `record` and field number `field`.

When creating the dBASE file with **xbase.new**, pass the `'f'` data type descriptor for the respective fields.

A proprietary extension, applications that import dBASE files - such as Microsoft Excel - do not support this type.

Note that 4-byte floats are inherently inaccurate, so when reading them back into Agena you will find significant round-off errors due to stray bits.

The return is **true** if writing succeeded, and **false** otherwise. Note that the return **false** only indicates that an error may have occurred.

`xbase.writedouble (filehandle, record, field, value)`

Writes the number `value` (4th argument) to the file denoted by `filehandle` to record number `record` and field number `field`.

When creating the dBASE file with **xbase.new**, pass the `'o'` (letter O) data type descriptor for the respective fields.

The number is stored in binary format of eight bytes (C double, converted to a C uint64_t). By default, doubles are written in Big Endian representation. You can change this to Little Endian by setting

```
> xbase.kernel(DoubleIsBigEndian = false);
```

A dBASE 7 extension, some applications that import dBASE files - such as Microsoft Excel - do not support binary numbers, but LibreOffice 5.x and beyond does.

The return is **true** if writing succeeded, and **false** otherwise. Note that the return **false** only indicates that an error may have occurred.

See also: **xbase.writefloat, xbase.writelong, xbase.writenumber, xbase.new**.

xbase.writefloat (filehandle, record, field, value)

Writes the number `value` (4th argument) to the file denoted by `filehandle` to record number `record` and field number `field`.

When creating the dBASE file with **xbase.new**, pass the `'F'` data type descriptor for the respective fields.

The number is stored with a total of 20 digits, including a maximum of 18 digits following the decimal point (scale).

The return is **true** if writing succeeded, and **false** otherwise. Note that the return **false** only indicates that an error may have occurred.

See also: **xbase.writedouble, xbase.writenumber, xbase.new**.

xbase.writelong (filehandle, record, field, value)

Writes the number `value` (4th argument) to the file denoted by `filehandle` to record number `record` and field number `field`.

When creating the dBASE file with **xbase.new**, pass the `'I'` data type descriptor for the respective fields.

The function automatically truncates Agena numbers containing decimal places to their integral part and issues an error if the numeric range [-2'147'483'647, +2'147'483'647] is exceeded.

By default, longs are written in Little Endian representation. You can change this to Big Endian by setting

```
> xbase.kernel(LongIsBigEndian = true);
```

A dBASE 7 extension, some applications that import dBASE files - such as Microsoft Excel - do not support binary numbers, but LibreOffice 5.x and beyond does.

The return is **true** if writing succeeded, and **false** otherwise. Note that the return **false** only indicates that an error may have occurred.

See also: **xbase.writedouble**, **xbase.writefloat**, **xbase.writenumber**, **xbase.new**.

`xbase.writenumber (filehandle, record, field, value)`

Writes the number `value` (4th argument) to the file denoted by `filehandle` to record number `record` and field number `field`.

The function automatically determines whether the respective field is of xBASE type Numeric (`'N'`), Float (`'F'`), binary Long (`'I'`), or Binary Double (`'O'`) (letter O). Concerning `'O'` and `'I'`, read the remarks on Endianness in the description of **xbase.writedouble**. It also writes the proprietary formats 'b' for byte (see **xbase.writebyte**) and 'c' for complex numbers (see **xbase.writecomplex**).

The return is **true** if writing succeeded, and **false** otherwise. Note that the return **false** only indicates that an error may have occurred.

See also: **xbase.writedouble**, **xbase.writefloat**, **xbase.writelong**, **xbase.new**.

`xbase.writestring (filehandle, record, field, value)`

Writes the string value (4th argument) to the file denoted by `filehandle` to record number `record` and field number `field`.

When creating the dBASE file with **xbase.new**, pass the `'C'` data type descriptor for the respective fields.

The return is **true** if writing succeeded, and **false** otherwise. Note that the return **false** only indicates that an error might have occurred.

`xbase.writetime (filehandle, record, field,`
`                  y, m, d [, h [, m [, s [, ms]]]])`

Writes the timestamp given by the year `y`, month `m`, day `d`, and optionally hour `h`, minute `mm`, second `s`, and milliseconds `ms` to the file denoted by `filehandle` to record number `record` and field number `field`.

By default, hours, minutes, seconds, and milliseconds default to 0. Milliseconds must be an integer in the range [0, 999].

When creating the dBASE file with **xbase.new**, pass the `'@'` data type descriptor for the respective fields.

The return is **true** if writing succeeded, and **false** otherwise. Note that the return **false** only indicates that an error might have occurred.

By default, the timestamp internally is represented by two signed 32-bit integers in Little Endian representation. You can change this to Big Endian representation by setting:

```
> xbase.kernel(LongIsBigEndian = true);
```

If the timestamp shall be represented by a binary double - the Julian Date, but the documentation available on dBASE Level 7 files is contradictory - you can set:

```
> xbase.kernel(TimestampIsDouble = true);
```

The endianness of the double is Big Endian by default. You can change this to Little Endian by setting:

```
> xbase.kernel(DoubleIsBigEndian = false);
```

See also: **xbase.writedate**.

12.4 ads - Agena Database System

As a *plus* package, this simple database is not part of the standard distribution and must be activated with the **import** statement, e.g. `import ads`.

Agena is a database for storing and accessing strings and currently supports three `base` types:

1. Sorted `databases` with a key and one or more values,

2. sorted `lists` which store keys only,

3. unsorted `sequences` to hold any value (but no keys).

With databases and lists, each record is indexed, so that access to it is very fast. If you store data with the same key multiple times in a database, the index points to the last record stored, so you always get a valid record.

Sequences do not have indexes, so searching in sequences is rather slow. However, all values can be read into the Agena environment rapidly (using **ads.getall**).

The Agena Database System (ADS) pays attention to both file size and fast I/O operation. To reduce file size, the keys (and values) are stored with their actual lengths (of C type `int32_t`, so keys and values can be of almost unlimited size) and they are not extended to a fixed standard length. To fasten I/O operations, the length of each key (and value) is also stored within the base file.

The following terms are used in this chapter:

Section	Description
header	various information on the data file, including the maximum number of possible records, the actual number of records, and the type of the base (database, list, or sequence).
index	only with databases and lists: area containing all file positions of the actual records. The index section is always sorted. Sequences do not contain an index section.
records	key-value pairs with databases, and keys with lists or sequences.

A sample session:

First activate the package:

```
> import ads alias
```

Create a new database (file `c:\test.agb`) including all administration data like number of records, etc.:

```
> createbase('c:/test.agb');
```

Open the database for access. The variable `fh` is the file handle which refers to the database file (`c:\test.agb`) and is used in nearly all ads functions.

```
> fh := openbase('c:/test.agb');
```

Put an entry into the database with key `Duck` and value `Donald`.

```
> writebase(fh, 'Duck', 'Donald');
```

Check what is stored for `Duck`.

```
> readbase(fh, 'Duck'):
Donald
```

Show information on the database:

```
> attrib(fh):
keylength ~ 31                     # Maximum length for key
type ~ 0                           # database type, 0 for relational database
stamp ~ AGENA DATA SYSTEM          # name of database
indexstart ~ 256                   # begin of index section in file
commentpos ~ 0                     # position of a description, 0 because none
                                   # was given.
version ~ 300                      # base version, here 3.00
maxsize ~ 20000                    # maximum number of possible records. Agena
                                   # automatically extends the database, if
                                   # this number is exceeded.
indexend ~ 80255                   # end of index section
creation ~ 2008/01/18-19:00:50     # number of creation
columns ~ 2                        # number of columns
size ~ 1                           # number of actual entries
```

Close the database. After that you cannot read or write any entries any longer. Use the **open** function if you want to have access again.

```
> closebase(fh);
```

On all ads database types, you may use the following procedures:

ads.attrib (filehandle)

Returns a table with all attributes of the `base` file. The table includes the following keys:

Key	Description	Type
'columns'	The number of columns in the base.	number
'commentpos'	The position of a comment in the base. If no comment is present, its value is 0.	number
'creation'	The date of creation of the base. The return is a formatted string including date and time.	string
'indexstart'	the first byte in the base file of the index section.	number
'indexend'	the last byte in the base file of the index section.	number
'keysize'	the maximum length of the record key.	number
'maxsize'	total number of data sets allowed.	number
'size'	the actual number of valid data sets (see **ads.sizeof** as a shortcut).	number
'stamp'	The base stamp at the beginning of the file.	string
'type'	Indicator for database (0), list (1), or sequence (2).	number
'version'	The base version.	number
'description'	The description, empty string if not present	string

If the file is not open, **attrib** returns **false**.

See also: **ads.free**, **ads.sizeof**.

ads.clean (filehandle)

Physically deletes all entries that have become invalid (i.e. replaced by new values) from the database or list. The file index section is adjusted accordingly and the file shrunk to the new reduced size.

If there are no invalid records, **false** will be returned. If all records could be deleted successfully, **true** will be returned. If the file is not open, the result is **fail**. If a file truncation error occurred, clean quits with an error. The function issues an error if the file contains a sequence.

ads.closebase (filehandle [, filehandle2, ···])

Closes the base(s) identified by the given file handle(s) and returns **true** if successful, and **false** otherwise. **false** will be returned if at least one base could not be closed. The function also deletes the file handles and the corresponding filenames from the ads.openfiles table.

```
ads.comment (filehandle)

ads.comment (filehandle, comment)

ads.comment (filehandle, '')
```

In the first form, the function returns the comment stored to the database or list if present. The return is a string or **null** if there is no comment.

In the second form, **ads.comment** writes or updates the given comment to the database or list and if successful, returns **true**. The comment is always written to the end of the file. If it could not successfully add or update a comment, the function quits with an error.

In the third form, by passing an empty string, the existing comment is entirely deleted from the database or list.

If `filehandle` points to a sequence, an error will be issued and no comment is written. **fail** will be returned, if the file is not open.

Internally, the position of the comment is stored in the file header. See ads.attrib ['commentpos'].

```
ads.createbase (filename
    [, number_of_records [, type [, number_of_columns
    [, length_of_key [, description]]]]])

ads.createbase (filename
    [, number_of_records [, type [, length_of_key [, description]]]])
```

Creates and initialises the index section of a new base with the given number of columns. It returns the file handle as a number, and closes the created file.

The first form defines a database, the second form is used to create sequences and lists.

Arguments / Options:

filename	The path and full name of the base file.
number_of_records	The maximum number of records in the base. Default is 20,000. If you pass 0, fail will be returned and the base is not created.
type	By default, the type is 'database'. If you pass the string 'list', then a list will be created. The string 'seq' will create a sequence. If the type passed is not known, **fail** will be returned and no base is created.

number_of_columns	The number of columns in a database. Default: 2 (key and value). If the base is not a database, do not pass any value (see second form). If the number of columns is non-positive, **fail** will be returned and no base will be created.
length_of_key	The maximum length of the base key. Note that internally, the length is incremented by 1 for the terminating \0 character. Default: 31 including the terminating \0 character.
description	A string with a description of the contents of the base. A maximum of 75 characters is allowed (including the \0 character). If the string is too long, it will be truncated. Default: 75 spaces.

`ads.createseq (filename)`

Creates a sequence with the given `filename` (a string). The function is written in Agena and can be used after issuing `import ads`.

`ads.desc (filehandle)`
`ads.desc (filehandle, description)`

In the first form, returns the description of a base stored in the file header. If no description has explicitly been written, the function returns the empty string.

In the second form, **ads.desc** sets or overwrites the description section of a database or list. Pass the description as a string. If the string is longer than 75 characters, **fail** will be returned and there are no changes to the base file. If the file is not open, **fail** will be returned, as well. If it was successful, the return is **true**.

`ads.expand (filehandle [, n])`

Increases the maximum number of datasets by `n` records (`n` an integer). By default, `n` is 10. Internally, all data sets are shifted, so that the index section in the data file can be extended. Thus, the greater `n`, the faster shifting will be if the function is called many times, which is significant for large files.

The function returns **fail** if the file is not open, and **true** otherwise. It issues an error if the file contains a sequence.

`ads.filepos (filehandle)`

Returns the current position of the file denoted by `filehandle`. See also: **ads.attrib**.

`ads.free (filehandle)`

Determines the number of free data sets and returns them as an integer. If the base has not open, it returns **fail**. See also: **ads.attrib**.

`ads.getall (filehandle [, option])`

Converts an ADS sequence to a set and returns this set. The function automatically initialises the set with the number of entries in the ADS sequence. If the file is not open, **fail** will be returned.

If any option is given, an Agena sequence will be returned with the entries in the order of their physical presence in the database file; if one and the same entry is stored multiple times, it is also returned multiple times in the sequence.

See also: **ads.getkeys**, **ads.getvalues**.

`ads.getkeys (filehandle)`

Gets all valid keys in a database or list and returns them in a table. Argument: file handle (integer). If the file is not open, **fail** will be returned. If the base is empty, **null** will be returned. The function issues an error if the file contains a sequence.

See also: **ads.get**, **ads.getvalues**.

`ads.getvalues (filehandle [, column])`

By default, gets all valid entries in the second column in a database and returns them in a table. If the optional argument `column` is given, the entries in this column are returned. Argument: file handle (integer). If the file is not open or if the column does not exist, **fail** will be returned. If the base is empty, **null** will be returned. With lists, the return is always **null**.

See also: **ads.get**, **ads.getkeys**.

`ads.index (filehandle, key)`

Searches for the given key (a string) in the base pointed to by filehandle and returns its file position as a number. If their are no entries in the set, the function returns **null**. If the file is not open, **fail** will be returned.

`ads.indices (filehandle)`

Returns the file positions of all valid datasets as a table.

If the file is not open, indices returns **fail**. If there are no entries in the base, the return is an empty table, otherwise a table with the indices will be returned. The function issues an error if the file contains a sequence.

See also: **ads.retrieve**, **ads.invalids**, **ads.peekin**, **ads.index**.

`ads.invalids (filehandle)`

Returns the file positions of all invalid records in a database as a table.

If the file is not open, invalids returns **fail**. If no invalid entries are found, the return is an empty table. See also **ads.retrieve**. Note that the function also works with lists. However, since lists never contain invalid records, an empty table will always be returned with lists.

With sequences, the function issues an error.

`ads.iterate (filehandle [, key])`

Iterates sequentially and in ascending order over all keys in the database or list. With databases, both the next key and its corresponding value are returned. With lists, only the next key is returned.

The very first `key` can be accessed with an empty string or **null** (or by only passing `filehandle`). If there are no more keys left, the function returns **null**. If the database is empty, **null** will be returned as well. If the file is not open, the function returns **fail**.

Example:

```
> s, t := ads.iterate(fh, '');
```

```
> s, t := ads.iterate(fh, s);
```

With ADS sequences, the function returns an iterator function that when called returns the next entry in it.

`ads.lock (filehandle)`
`ads.lock (filehandle, size)`

The function locks the file given by its handle `filehandle` so that it cannot be read or overwritten by other applications.

In the first form, the entire file is locked in UNIX-based systems. In Windows, only 2^{63} bytes are locked, so you have to use the second form in Windows after the file has become larger than 2^{63} bytes (= 8,589,934,592 GBytes).

In the second form the function locks `size` bytes from the current file position. Locked blocks in a file may not overlap. `size` may be larger than the current file length.

Note that other applications that do not use the locking protocol may nevertheless have read and write access to the file.

See also: **ads.unlock**.

`ads.openbase (filename [, anything])`

Opens the base with name filename and returns a file handle (a number). If it cannot find the file, or the base has not the correct version number, the function

returns **fail**. The base is opened in both read and write mode.

If an optional second argument is given (any valid Agena value), the base is opened in read mode only.

The function also enters the newly opened file into the ads.openfiles table.

`ads.openfiles`

A global table containing all files currently open. Its keys are the file handles (integers), the values the file names (strings). If there are no open files, **ads.openfiles** is an empty table.

`ads.peekin (filehandle, position)`

Returns both the length of an entry (including the terminating \0 character) and the entry itself at the given file position as two values (an integer and a string). The function is save, so if you try to access an invalid file position, the function will exit returning **fail**. It issues an error if the file contains a sequence.

See also: **ads.index**, **ads.retrieve**.

`ads.rawsearch (filehandle, key [, column])`

With databases, the function searches all entries in the given column for the substring key and returns all respective keys and the matching entries in a table. If column is omitted, the second column is searched. The value for column must be greater than 0, so you can also search for keys.

With lists and sequences, the function always returns **null**. If the base is empty, **null** will be returned.

If the file is not open or the column does not exist, the function returns **fail**.

See also: **ads.read**, **ads.getvalues**.

`ads.readbase (filehandle, key)`

With databases, the function returns the entry (a string) to the given key (also a string). With lists and sequences, the function returns **true** if it finds the key, and **false** otherwise.

If the file is not open, read returns **fail**. If the base is empty, **null** will be returned. The function uses binary search.

See also **ads.rawsearch**.

`ads.remove (filehandle, key)`

With databases, the function deletes a key-value pair from the database; with lists, the key is deleted. Physically, only the key to the record is deleted, the key or key-value pair still resides in the record section but cannot be found any longer.

The function returns **true** if it could delete the data set, and **false** if the set to be deleted was not found. If the file is not open, delete returns fail. The function issues an error if the file contains a sequence.

If you want to physically delete all invalid records, use **ads.clean**.

`ads.retrieve (filehandle, position)`

Gets a key and its value from a database or list (indicated by its first argument, the file handle) at the given file position (an integer, the second argument). Two values are returned: the respective key and its value. With lists, only the key will be returned.

The function is save, so if you try to access an invalid file position, the function will exit and return **fail**.

If the file is not open, retrieve returns **fail.** The function issues an error if the file contains a sequence.

See also **ads.indices**, **ads.invalids**.

`ads.sizeof (filehandle)`

Returns the number of valid records (an integer) in the base pointed to be filehandle. If the base pointed to by the numeric filehandle is not open, the function returns **fail**.

`ads.sync (filehandle)`

Flushes all unwritten content to the base file. The function returns **true** if successful, and **fail** otherwise (e.g. if the file was not opened before or an error during flushing occurred).

`ads.unlock (filehandle)`

`ads.unlock (filehandle, size)`

The function unlocks the file given by its handle `filehandle` so that it can be read or overwritten by other applications again. For more information, see **ads.lock**.

`ads.writebase (filehandle, key [, value1, value2, ···])`

With databases, the function writes the key (a string) and the values (strings) to the database file pointed to by filehandle (an integer). If value is omitted, an empty string is written as the value.

With lists, the function writes only the key (a string) to the database file. If you pass values, they are ignored. If the key already exists, nothing is written or done and **true** will be returned. Thus, lists never contain invalid records.

In both cases, the index section is updated. If a key already exists, its position in the index section is deleted and the new index position is inserted instead (in this case there is no re-shifting). This does not remove the actual key-value pair in the record section. The function always writes the new key-value pair to the end of the file. (The file position after the write operation has completed is always 0.)

If the maximum number of possible records is exceeded, the base is automatically expanded by 10 records. You do not need to do this manually.

write returns the **true** if successful. If the file is not open, **write** returns **fail**.

12.5 xml - XML Parser

As a *plus* package, the **xml** package is not part of the standard distribution and must be activated with the **import** statement, e.g. `import xml`. It is available for Solaris, OS/2, DOS, Mac OS X, Linux, and Windows only.

Since the XML package actually is the LuaExpat binding with some few Agena-specific modifications, large portions of this subchapter have been taken from the LuaExpat documentation.

12.5.1 Introduction

XML/LuaExpat is a SAX XML parser based on the Expat library. SAX is the Simple API for XML and allows programmes to:

- process a XML document incrementally, thus being able to handle huge documents without memory penalties;

- register handler functions which are called by the parser during the processing of the document, handling the document elements or text.

With an event-based API like SAX the XML document can be fed to the parser in chunks, and the parsing begins as soon as the parser receives the first document chunk. XML/LuaExpat reports parsing events (such as the start and end of elements) directly to the application through callbacks. The parsing of huge documents can benefit from this piecemeal operation.

XML/LuaExpat is distributed as a library.

12.5.2 Parser objects

Usually SAX implementations base all operations on the concept of a parser that allows the registration of callback functions. XML/LuaExpat offers the same functionality but uses a different registration method, based on a table of callbacks.

This table contains references to the callback functions which are responsible for the handling of the document parts. The parser will assume no behaviour for any undeclared callbacks.

12.5.3 Shortcuts

`xml.decode (str)`

Reads a string `str` containing an XML stream and converts it into a dictionary. Its return is rather raw, but it can cope with situations where one and the same XML object is present multiple times on the same hierarchy.

`xml.decodexml (str)`

Reads a string `str` containing an XML stream and converts it into a dictionary.

The function provides some checking (basic syntax and balanced tags), and supports namespaces, XML and DOCTYPE declarations, comments and processing instructions. If a XML tag includes hyphens or colons, then they are converted to underscores in the corresponding Agena dictionary key.

The data must be included in an envelope.

The function also returns processing instructions in the `xattr` tag.

The function is written in Agena and included in the lib/xml.agn file.

The function does not cope well if one and the same XML object is present multiple times on the same hierarchy. Use **utils.decodexml** or **xml.decode** instead.

`xml.readxml (filename)`

Reads an XML file and returns its data in an Agena dictionary. The data must be included in an envelope.

See also: **utils.readcsv**, **utils.readxml**, **xml.decode**, **xml.decodexml**.

12.5.4 Constructor

`xml.new (callbacks [, separator])`

The parser is created by a call to the function xml.new, which returns the created parser or raises a Lua error. It receives the callbacks table and optionally the parser separator character used in the namespace expanded element names.

12.5.5 Functions

`xml.close (parser)`

Closes the parser, freeing all memory used by it. A call to close(parser) without a previous call to parse(parser) could result in an error.

`xml.getbase (parser)`

Returns the base for resolving relative URIs.

`xml.getcallbacks (parser)`

Returns the callbacks table.

```
xml.parse (parser, s)
```

Parse some more of the document. The string s contains part (or perhaps all) of the document. When called without arguments the document is closed (but the parser still has to be closed).

The function returns a non **null** value when the parser has been successful, and when the parser finds an error it returns five results: **null**, msg, line, col, and pos, which are the error message, the line number, column number and absolute position of the error in the XML document.

```
xml.pos (parser)
```

Returns three results: the current parsing line, column, and absolute position.

```
xml.setbase (parser, base)
```

Sets the base to be used for resolving relative URIs in system identifiers.

```
xml.setencoding (parser, encoding)
```

Sets the encoding to be used by the parser. There are four built-in encodings, passed as strings: 'US-ASCII', 'UTF-8', 'UTF-16', and 'ISO-8859-1'.

12.5.6 Callbacks

The Agena callbacks define the handlers of the parser events. The use of a table in the parser constructor has some advantages over the registration of callbacks, since there is no need for the API to provide a way to manipulate callbacks.

Another difference lies in the behaviour of the callbacks during the parsing itself. The callback table contains references to the functions that can be redefined at will. The only restriction is that only the callbacks present in the table at creation time will be called.

The callbacks table indices are named after the equivalent Expat callbacks:

CharacterData, Comment, Default, DefaultExpand, EndCDataSection, EndElement, EndNamespaceDecl, ExternalEntityRef, NotStandalone, NotationDecl, ProcessingInstruction, StartCDataSection, StartElement, StartNamespaceDecl, and UnparsedEntityDecl.

These indices can be references to functions with specific signatures, as seen below. The parser constructor also checks the presence of a field called _nonstrict in the callbacks table. If _nonstrict is absent, only valid callback names are accepted as indices in the table (Defaultexpanded would be considered an error for example). If _nonstrict is defined, any other fieldnames can be used (even if not called at all).

The callbacks can optionally be defined as **false**, acting thus as placeholders for future assignment of functions.

Every callback function receives as the first parameter the calling parser itself, thus allowing the same functions to be used for more than one parser for example.

`callbacks.CharacterData = proc(parser, string)`

Called when the parser recognises an XML CDATA string.

`callbacks.Comment = proc(parser, string)`

Called when the parser recognises an XML comment string.

`callbacks.Default = proc(parser, string)`

Called when the parser has a string corresponding to any characters in the document which wouldn't otherwise be handled. Using this handler has the side effect of turning off expansion of references to internally defined general entities. Instead these references are passed to the default handler.

`callbacks.DefaultExpand = proc(parser, string)`

Called when the parser has a string corresponding to any characters in the document which wouldn't otherwise be handled. Using this handler doesn't affect expansion of internal entity references.

`callbacks.EndCdataSection = proc(parser)`

Called when the parser detects the end of a CDATA section.

`callbacks.EndElement = proc(parser, elementName)`

Called when the parser detects the ending of an XML element with elementName.

`callbacks.EndNamespaceDecl = proc(parser, namespaceName)`

Called when the parser detects the ending of an XML namespace with namespaceName. The handling of the end namespace is done after the handling of the end tag for the element the namespace is associated with.

`callbacks.ExternalEntityRef = proc(parser, subparser, base, systemId, publicId)`

Called when the parser detects an external entity reference.

The subparser is a XML/LuaExpat parser created with the same callbacks and Expat context as the parser and should be used to parse the external entity.

The base parameter is the base to use for relative system identifiers. It is set by setbase and may be **null**.

The systemId parameter is the system identifier specified in the entity declaration and is never **null**.

The publicId parameter is the public id given in the entity declaration and may be **null**.

```
callbacks.NotStandalone = proc(parser)
```
Called when the parser detects that the document is not `standalone`. This happens when there is an external subset or a reference to a parameter entity, but the document does not have standalone set to "yes" in an XML declaration.

```
callbacks.NotationDecl =
    proc(parser, notationName, base, systemId, publicId)
```
Called when the parser detects XML notation declarations with notationName.

The base parameter is the base to use for relative system identifiers. It is set by setbase and may be **null**.

The systemId parameter is the system identifier specified in the entity declaration and is never **null**.

The publicId parameter is the public id given in the entity declaration and may be **null**.

```
callbacks.ProcessingInstruction = proc(parser, target, data)
```
Called when the parser detects XML processing instructions. The target is the first word in the processing instruction. The data is the rest of the characters in it after skipping all whitespace after the initial word.

```
callbacks.StartCdataSection = proc(parser)
```
Called when the parser detects the begining of an XML CDATA section.

```
callbacks.StartElement = proc(parser, elementName, attributes)
```
Called when the parser detects the begining of an XML element with elementName.

The attributes parameter is a table with all the element attribute names and values. The table contains an entry for every attribute in the element start tag and entries for the default attributes for that element.

The attributes are listed by name (including the inherited ones) and by position (inherited attributes are not considered in the position list).

As an example if the book element has attributes author, title and an optional format attribute (with `printed` as default value),

    <book author=\"Ierusalimschy, Roberto\" title=\"Programming in Lua\">

would be represented as

```
[1 ~ 'author',
 2 ~ 'title',
 author ~ 'Ierusalimschy, Roberto',
 format ~ 'printed',
 title ~ 'Programming in Lua']
```

callbacks.StartNamespaceDecl = proc(parser, namespaceName)

Called when the parser detects an XML namespace declaration with namespaceName. Namespace declarations occur inside start tags, but the StartNamespaceDecl handler is called before the StartElement handler for each namespace declared in that start tag.

callbacks.UnparsedEntityDecl =
  proc(parser, entityName, base, systemId, publicId, notationName)

Called when the parser receives declarations of unparsed entities. These are entity declarations that have a notation (NDATA) field.

As an example, in the chunk

    <!ENTITY logo SYSTEM "images/logo.gif" NDATA gif>

entityName would be "logo", systemId would be "images/logo.gif" and notationName would be "gif". For this example the publicId parameter would be **null**. The base parameter would be whatever has been set with setbase. If not set, it would be **null**.

The separator character:

The optional separator character in the parser constructor defines the character used in the namespace expanded element names. The separator character is optional (if not defined the parser will not handle namespaces) but if defined it must be different from the character '\0'.

12.6 tar - UNIX tar

As a *plus* package, the tar package is not part of the standard distribution and must be activated with the **import** statement, e.g. `import tar`.

12.6.1 Introduction

This package lists, reads, and extracts individual files from a UNIX tar archive.

See also: **gzip** package.

12.6.2 Functions

`tar.close (fh)`

Closes an archive archived file denoted by its file handle `fh` and returns **true** on success and **false** otherwise.

The function is written in Agena (see lib/tar.agn).

`tar.extract (fn [, pattern])`

Extracts files, directories, and symbolic links from the given tar archive `fn`, a file name of type string to the given current working directory. By default, all files are extracted. If a second argument `pattern` is given, then only the files matching the given pattern - a string - are copied. `pattern` may include wildcards, see **strings.glob**.

The return is a table of all the files extracted.

The function is written in Agena (see lib/tar.agn).

`tar.lines (fh, length)`

Creates an iterator function that with each call returns a new line of a file included in a tar file. The `length` (in bytes) of the archived file pointed to by `fh` must be given as the second argument.

`fh` is a numeric file handle returned by calling **tar.open**. Since **tar.open** also returns the length as a second return, it can be easily passed to **tar.lines**.

If the end of the archived file has been reached, the iterator function returns **null**. The iterator does not close the file connection, use **tar.close** to accomplish this.

The function is written in Agena (see lib/tar.agn).

`tar.list (fn [, pattern])`

Returns all files in the UNIX tar file fn (a file name), and returns a table of tables with the following information:

* file name (key `'name'`),
* file mode (key `'mode'`),
* start position (key `'start'`, expressed as the offset to the beginning of the file),
* file length in bytes (key `'length'`),
* file timestamp in UNIX time (key, `'timestamp'`, decimal number of seconds since the start of a given epoch, use `os.date` to convert it into calendar date/time),
* ustar indicator (`'ustar'` if set, else the empty string),
* numeric owner id (key `'ownerid'`, decimal),
* numeric group id (key `'groupid'`, decimal),
* and the decimal checksum (key `'checksum'`).

If a second argument `pattern` is given, then only the files matching the given pattern - a string - are returned. `pattern` may include wildcards, see **strings.glob**.

The function is written in Agena (see lib/tar.agn).

`tar.open (tarfile, fn)`

Opens an archived file `fn` (a file name) in the tar file given by `tarfile` (also a file name), sets the file pointer to the beginning of the actual contents of the archived file (i.e. not its tar header), and returns both a numeric file handle to the archived file and its size.

The function is written in Agena (see lib/tar.agn).

12.7 gzip - Library to Read and Write UNIX gzip Compressed Files

As a *plus* package, in Solaris, Linux, Mac OS X, OS/2, DOS, and Windows, this library is not part of the standard distribution and must be activated with the **import** statement, e.g. `import gzip`. See also: **tar** package.

A typical session may look like this:

```
> import gzip;

> fd := gzip.open('primes.dat.gz', 'r'):
gzipfile(0096A9F8)

>for keys I in gzip.lines(fd) do print(i) od;

> gzip.close(f):
true
```

gzip.close (filehandle [, filehandle, ···])

Closes the files denoted by the given file handles.

gzip.deflate (str)

The function compresses the string `str`. The return is the compressed string and its size, in this order.

See also: **gzip.inflate**.

gzip.flush (filehandle)

This function takes a file handle and flushes all output to the working file.

gzip.inflate (str [, n])

The function decompresses a inflated string `str`. The size of the original uncompressed string is given as the second argument `n`, which by default is size(`str`) + 16383)/16384. The return is the uncompressed string.

See also: **gzip.deflate**.

`gzip.lines (filehandle)`

`gzip.lines (filename)`

Returns an iterator function that, each time it is called, returns a new line from the file. Therefore, the construction

```
  for keys line in gzip.lines(file) do ... od
```

will iterate over all lines of the file.

If a file name is given, the file is closed when the loop ends. If a file handle is given, the file is not closed.

`gzip.open (filename [, mode])`

Opens a file name. If mode is not given, a default mode `'rb'` will be used. `mode` can include special modes such as characters `'1'` to `'9'` that will be treated as the compression level when opening a file for writing.

It returns a new file handle, or, in case of errors, **null** plus an error message.

`gzip.read (filehandle, format`$_1$`, ···)`

Reads the file with the given file handle, according to the given formats, which specify what to read. For each format, the function returns a string with the characters read, or **null** if it cannot read data with the specified format. When called without formats, it uses a default format that reads the entire next line (see below).

The available formats are:

- `'*a'` reads the whole file, starting at the current position. On end of file, it returns the empty string.
- `'*l'` reads the next line (skipping the end of line), returning **null** on end of file. This is the default format.
- number reads a string with up to that number of characters, returning **null** on end of file. If number is zero, it reads nothing and returns an empty string, or **null** on end of file.

Unlike **io.read**, the `'*n'` format is not available.

`gzip.seek (filehandle [, whence] [, offset])`

Sets and gets the file position, measured from the beginning of the file, to the position given by `offset` plus a base specified by the string `whence`, as follows:

- 'set' base is position 0 (beginning of the file),
- 'cur' base is current position,
- 'end' is the end of the file.

In case of success, **seek** returns the final file position, measured in bytes from the beginning of the file. If this function fails, it returns **null**, plus a string describing the error.

The default value for `whence` is 'cur', and for `offset` is 0. Therefore, the call gzip.seek(filehandle) returns the current file position, without changing it; the call gzip.seek(filehandle, 'set') sets the position to the beginning of the file (and returns 0); and the call gzip.seek(filehandle, 'end') sets the position to the end of the file, and returns its size.

`gzip.write (filehandle, value₁, ···)`

Writes the value of each of its arguments to the file specified by `filehandle`. The arguments must be strings or numbers. To write other values, use **tostring** or **strings.format** before **write**.

Chapter **Thirteen**

Communication

13 Communication

13.1 net - Network Library

As a *plus* package, in Solaris, Linux, Mac OS X, and Windows, this library is not part of the standard distribution and must be activated with the **import** statement, e.g. `import net`.

13.1.1 Introduction and Examples

This package provides basic functions to pass text from a client to a server using the IPv4 protocol. Thus it is suited to exchange information over the Internet and Local Area Networks.

> **Please remember that the package only supports unencrypted data transfer which might be insecure ! There is no SSL support.**
>
> If you do not use this package, no network functionality will be activated.
>
> **Please also note that when using *net.accept*, *net.connect*, *net.receive*, *net.send*, and *net.survey*, you will give access to your computer through LANs or the Internet, so please programme handshaking and blacklist/whitelist methods.**
>
> Limited white and blacklisting to allow or prohibit connections is supported through the ***net.whitelist*** and ***net.blacklist*** feature.

Communication is performed with `stream sockets` that ensure that data is sent and received in the original order and hopefully without errors. A socket is being created by a call to the **net.open** function.

In the following example, we will set up a one-way communication with the `client` sending and the `server` receiving data.

A typical session might begin by setting up the server. This is because a client cannot connect to a server until the latter is ready for it.

```
> import net alias
net v0.2.1 as of January 13, 2013

accept,  address,  bind,  block,  close,  connect,  listen,  lookup,  open,
opensockets, receive, remoteaddress, send, shutdown, survey
```

Create a socket: the **net.open** function returns a new socket handle:

```
> s := open():
932
```

Now associate this socket with a port on the server machine[24] by running **net.bind**. In this example we expect data to be received on your own computer on port 1300.

```
> bind(s, '127.0.0.1', 1300):
127.0.0.1   1300
```

Now our socket must be converted to a server socket by calling

```
> listen(s):
true
```

and be told to get a pending connection by running **net.accept**.

net.accept waits until a client asks the server for a connection (see client example below). It returns a new socket handle which later on manages this specific connection, while the original socket is ready to wait for requests for other connection.

net.accept also returns the IP address of the client asking for a connection, and its port.

```
> t, ip, port := accept(s):
924    127.0.0.1    3230
```

If you do not want **net.accept** to wait indefinitely until something happens, call **net.block** with **the original server socket and false** as its second argument.

Please note that you should check the incoming connection against a white or black list so that only trusted clients can send you any data. To decline and terminate an incoming connection, either check the incoming caller and just call **net.close** with the handle returned by **net.access**, or use the built-in basic black and whitelist functionality described at the end of this subchapter.

It also a good idea to validate the incoming connection with a handshaking procedure which checks the incoming data for certain information and then automatically decides whether to go on or shut down the connection.

Data received from the client will be returned by calling **net.receive** with the new file handle returned by **net.accept**.

```
> receive(t):
Kuckuck !  9
```

Finally, close both sockets (or just the handle returned by **net.accept**):

```
> close(t, s):
true
```

---

[24] You may use the operating system commands `ifconfig` (UNIX, Mac) or `ipconfig` (Windows) to determine your own IP address.

To open a client session, start Agena in another shell:

```
> import net alias
```

To connect to a server, first issue:

```
> d := open()
932
```

Now connect to the server by passing the socket handle, the IP address and port number of the server. `'localhost'` means that the server runs on the same machine as the client.

```
> connect(d, 'localhost', 1300):
true
```

Send some text once or more.

```
> send(d, 'Kuckuck !'):
9
```

The server immediately returns the text sent. To finish a client session, type:

```
> close(d):
true
```

Call **net.opensockets** to have a look at the state of all open sockets.

Following now is an extended but crude example for a one-way connection which sends one thousand hashes from the client to the server on the local host on port 1300.

Since with one single call, **net.receive** by default processes `only` 512 bytes in Windows and usually 8,192 bytes in UNIX, the server uses a **while** loop to receive all the data until the client closes the connection.

Since **net.receive** returns two results - the string and the number of characters received - its second return will be 0 if the client terminates a network session.

Server	Client

<pre>> import net alias
 	
> d := open():
132
 	
> bind(d, 'localhost', 1300):
127.0.0.1   1300
 	
> listen(d):
true
 	
> e, f, g := accept(d);
 	
> print(e,f, g);
352   127.0.0.1   49178
 	
> x, y := receive(e);
 	
> print(x, y);
##### (512 hashes) ##### 512 	
 	
> while y <> 0 do
>   x, y := receive(e);
>   print(x, y);
> od;
##### (more hashes) #### 488 	
      0
 	
> close(e, d):
true</pre>
	<pre>> import net alias
	
	> d := open():
	352
	
	> connect(d, 'localhost', 1300):
	true
	
	> send(d, strings.repeat('#', 1m)):
	1000000
	
	> close(d):
	true</pre>

A simple bi-directional connection:

Server	Client

<pre>> import net alias
 	
> d := open():
124
 	
> bind(d, 'localhost', 1300):
127.0.0.1   1300
 	
> listen(d):
true
 	
> e, f, g := accept(d);
 	
> print(e,f, g);
344   127.0.0.1   49183
 	
> x, y := receive(e);
 	
> print(x, y);
## etc. 512 	
 	
> send(e, 'Got ' & y & ' bytes');</pre>
	<pre>> import net alias
	
	> d := open():
	124
	
	> connect(d, 'localhost', 1300):
	true
	
	> send(d, strings.repeat('#', 1k)):
	1000
	
	> receive(d):
	Got 512 bytes 13
	
	> receive(d):
	Got 488 bytes 13
	
	> close(d):
	true</pre>

Server	Client
| ```
> while y <> 0 do
>    x, y := receive(e);
>    print(x, y);
>        send(e,  'Got  '  &  y  &  '
bytes');
> od;
## etc.          488
        0

> close(e, d):
true
``` | |

Usage of black and whitelists: First initialise the **net** package.

```
> import net alias
```

Now put one or more a numeric (!) IPs to be blocked into the set **net.blacklist** to prohibit connections to these addresses (valid for both **net.connect** and **net.accept**).

```
> net.blacklist := {'127.0.0.1'}

> d := open():
3

> connect(d, '127.0.0.1', 1300):
Error in `net.connect`: partner in blacklist, closing socket 3.

Stack traceback: in `connect`
   stdin, at line 1 in main chunk
```

Socket d is now closed:

```
> opensockets():
[]
```

Now define a whitelist with all IPs to which a connection is allowed.

```
> net.whitelist := {'127.0.0.2'}

> d := open():
3

> return connect(d, '127.0.0.3', 1300)
Error in `net.connect`: partner not in whitelist, closing socket 3.

Stack traceback: in `connect`
   stdin, at line 1 in main chunk
```

The socket is closed, as well.

```
> opensockets():
[]
```

13.1.2 Functions

`net.accept (s)`

Accepts a connection request from a client on the given server socket handle `s`. If the server socket has been set to blocking mode, it waits until there is an incoming connection.

The function returns a new socket handle (a number) for the data to be received later on, and the address (a string) and port (a number) of the client socket.

Please note that the new socket created by **net.accept** must be closed separately to avoid too many open sockets.

The function also checks the global sets **net.blacklist** and **net.whitelist**, in this order, and if they exist. If you are trying to accept a connect from an address that is included in **net.blacklist**, then **net.accept** refuses this connection, closes the new socket that it created (see above), and issues an error. If you are trying to accept a connection from an address that is not in **net.whitelist**, the function does not establish a connection, closes the freshly created socket, and issues an error, as well.

Please note that **net.blacklist** and **net.whitelist must only contain <u>numeric</u> IPs**, and not addresses like 'sunsite.abc.xyz'. However, **net.accept** tries to convert the incoming address to a numeric IP address and then checks both lists[25]. If an address could not be resolved, the function does not allow a connection, and closes the newly created socket, and finally issues an error.

You may use **protect** in order to intercept the errors described above, but you must take care yourself for allowing or prohibiting a connection.

You have to set up **net.blacklist** and/or **net.whitelist** yourself after initialising the **net** package.

The procedure is a binding to C's `accept` function.

See also: **net.accept**, **net.bind**, **net.block**, **net.listen**, **net.receive**, **net.survey**.

---

[25] Usually, the server that tries to connect sends its numeric IP address, but probably it does not. So this is just a precautionary action.

`net.admin`

Table containing various operating system-specific administrative network settings:

Key	Meaning
`maxnsockets`	estimated maximum number of open sockets allowed
`protocols`	a table containing the supported protocols

`net.address (s)`

Returns two values: the IP address (a string) and port number (a number) to which socket `s` is bound.

See also: **net.lookup**, **net.remoteaddress**.

`net.bind (s [, address [, port]])`

Associates a socket `s` with an IP address and a port on the local machine and returns its IP address (a string) and the respective port on success or returns **false** and a string containing the error message otherwise.

If `address` is not given, localhost is bound to the socket (i.e. your own computer), otherwise the numeric IP address or host name is bound.

By default, port 1234 is connected, but you may specify another port (an integer) as a third argument. This might require administrative rights.

The procedure is a binding to C's `bind` function.

To determine your own IP address, open a shell and issue the command `ipconfig` in Windows, and `ifconfig` in Solaris, Linux, Mac, or other UNIX based platforms.

See also: **net.accept**, **net.listen**, **net.receive**, **net.survey**.

`net.block (s, mode)`

Sets a socket to blocking or non-blocking mode. The functions expects the socket handle (a number) `s` as its first argument and the `mode` (a Boolean) as its second argument. If the second argument is **true**, the socket is set to blocking mode, else to non-blocking mode. The return is **true** on success and **false** otherwise.

The procedure is a binding to C's `fcntl` (UNIX) or `ioctlsocket` (Windows) function.

`net.close (···)`

Terminates all the *given* servers or clients denoted by their socket handles and returns **true** on success, or **false** and a string containing an error message otherwise.

The procedure is a binding to C's `close` or `closesocket` function.

net.closewinsock ([anything])

The function is available only in the Windows edition. It finally terminates the current network session and returns **true** on success, or issues an error otherwise if `anything` is not given. If any value `anything` is passed to the function, in case of an error it returns **fail** plus an error message of type string.

Please note that when you call this function, no further network communication will be possible. Call **net.openwinsock** to enable network communication again.

The procedure is a binding to C's `WSACleanup` function.

See also: **net.openwinsock**.

net.connect (s [, address [, port]])

Connects the client denoted by it socket handle `s` (first argument, a number) to a server at the specified IP `address` (second argument, a string) and its `port` (third argument) so that data can be sent later. If `address` is missing, the address is set to `'localhost'`, if `port` is missing, port 1234 will be used.

If the client socket is set to blocking mode, the function waits until the server responds; if the client socket is set to non-blocking mode, it immediately returns without waiting for a server response.

The return is either **true** in case of success or **false** and the error message (a string) at failure.

The function also checks the global sets **net.blacklist** and **net.whitelist**, in this order, and if they exist. If you are trying to connect to an address that is included in **net.blacklist**, then **net.connect** does not establish a connection, closes socket `s`, and issues an error. If you are trying to connect to a server that is not in **net.whitelist**, the function does not establish a connection, closes the socket, and issues an error, as well.

Please note that **net.blacklist** and **net.whitelist must only contain <u>numeric</u> IPs**, and not addresses like 'sunsite.abc.yz'. However, **net.connect** tries to convert `address` to a numeric IP address and then checks both lists. If an address could not be resolved, the function does not establish a connection, closes socket `s` and issues an error.

You may use **protect** in order to intercept the errors described above, but you must take care yourself for allowing or prohibiting the connection.

You have to set up **net.blacklist** and/or **net.whitelist** yourself after initialising the **net** package.

The procedure is a binding to C's `connect` function.

See also: **net.send**.

net.isconnected ()

In Windows, checks whether you are currently connected to the internet and returns **true** or **false**. The function is not available on other platforms.

net.listen (s [, length])

Converts the given socket `s` to a server socket, enabling it to accept connections. You may optionally pass an integer in the range [1, 1024] determining the length of the queue for pending connections.

The return is either **true**, or **false** and a string with an error message if listening failed.

You must first run this function before calling **net.accept** and **net.receive**.

The procedure is a binding to C's `listen` function.

net.lookup ([x])

Determines the IP, an optional alias, the official name and the supported protocol of a given URL or numeric IP `x` of type string. If no argument is passed, the function will return the information on `'localhost'`.

An example:

```
> lookup('www.zeit.de'):
[networkaddress ~ [0.0.0.1], alias ~ [zeit.de], official ~ Die Zeit, type ~
IPv4]

> lookup('10.137.0.1'):
[networkaddress ~ [10.137.0.1], alias ~ [anything.yz], official ~ Anything,
type ~ IPv4]
```

See also: **net.address**, **net.remoteaddress**.

net.open ([blocking])

Creates a (client) network socket. If the optional first argument `blocking` is set to **false**, the socket is set to non-blocking mode.

The return is the socket handle (a number), the default address `'localhost'` and default port `1234`, the protocol (a number) and a Boolean indicating whether the handle can be reused by the system after the socket has been closed. If a new socket could not be opened, an error is issued.

net.open does not connect the client to a server - use **net.connect** for this.

To create a server socket waiting for input, use **net.bind**, **net.listen**, and **net.accept.**

The procedure is a binding to C's `socket` function.

See also: **net.close**.

net.opensockets ()

Returns all open sockets along with their respective attributes.

The return is a table with its keys the open socket handles, and their entries tables containing information on whether the socket is a server or client (key `'server'`, **true** or **false**), their own address (key `'address'`, a string), their own port (key `'port'`, a number), the protocol being used (key `'protocol'`, a number), whether the socket works in blocking or non-blocking mode (key `'blocking'`, **true** or **false**), and whether the socket has been connected to a server (`'connected'`, **true** or **false**).

The table key `'mode'` holds information on the read and write status of the socket:

Value	Meaning
`'none'`	the socket is not connected
`'shutdown'`	the socket no longer can receive or send data
`'read'`	the socket can only receive data, but cannot send any
`'write'`	the socket can only send data, but cannot receive any
`'readwrite'`	the socket can both send and receive data (the default)

Please note that modifying the contents of the table returned will not have any effect on the status of the sockets, so you cannot do any harm.

See also: **net.shutdown**.

net.openwinsock ([anything])

The function is available only in the Windows edition. It re-enables network communication and returns **true** on success, or issues an error otherwise if `anything` is not given. If any value `anything` is passed to the function, in case of an error it returns **fail** plus an error message of type string.

When initialising the **net** package by calling **readlib** or **with**, Agena automatically starts the Winsock daemon, so you do not have to call this function explicitly.

The procedure is a binding to C's `WSAStartup` function.

See also: **net.closewinsock**.

`net.receive (s [, getall [, maxlength]])`

Allows a server socket `s` to receive a string from a client. The function returns this string and its length (a number). `s` should be the socket handle returned by **net.accept**.

If the return is the empty string plus the value 0 (zero) for its length, the client has closed the connection - this is also a proper check on whether a client is still connected with a server socket. Please note that in this case, no further data can be received on this socket and you have to close `s` manually.

If **true** has been passed for the optional argument `getall`, the function reads in all data from the client until the latter closes the connection. If the client does not close the connection, **net.receive** waits infinitely.

The optional argument `maxlength` determines the maximum number of characters to be received. If a client tries to send more data than specified by `maxlength`, the function returns **false** and the string `'too many bytes received'`.

The maximum number of bytes to be read by one stroke is determined by **environ.kernel('buffersize')** which value depends on the operating system and can also mbe changed.

If any error occurs during receipt of the data, **net.receive** does not close the socket `s`, but returns **false** and a string containing either the message `'failure during receipt'` or `'too many bytes received'`, the latter if `maxlength` and the number of bytes received exceeded it.

The procedure is an extended binding to C's `recv` function.

See also: **net.accept**, **net.bind**, **net.block**, **net.listen**, **net.receive**, **net.send**, **net.survey**.

`net.remoteaddress (s)`

Returns two values: the IP address (a string) and port (a number) of the server that the client socket `s` is connected to.

See also: **net.address**, **net.lookup**.

`net.send (s, str [, true])`

Sends a string `str` (second argument) from the client denoted by its socket handle `s` (first argument, a number) to a server.

The return is the number of the characters actually sent. If the kernel decides not to send all the data in one chunk, the function might not send the complete string. If an optional third argument, the Boolean **true**, is given, **net.send**, however, tries to make sure that the complete string has been sent when it returns.

If `str` is the empty string, it will not be sent to the server.

The function returns **fail** and the string `'socket not connected'` if the socket has not been connected before by either **net.connect** or **net.accept**. It also returns fail and `'socket not connected'` if the connection has been disconnected.

If the number of bytes actually sent is not equal to the length of the string str, the function returns false, the string `'transfer size mismatch'`, and the number of bytes sent.

The procedure is an extended binding to C's `send` function.

See also: **net.connect**, **net.receive**.

net.shutdown (s, what)

The function stops further sends and receives on a socket `s`. If `what` is the string `'read'`, then the socket can no longer receive data; if `what` is the string `'write'`, it can lo longer send data; and if `what` is the string `'readwrite'`, it will not do both any longer.

Please note that socket `s` will still be active. Call **net.close** if you want to release the socket completely.

See also: **net.opensockets**.

net.smallping (ip, port [, iters [, delay [, message [, noprint]]]])

Opens a socket, connects to a server given by the string `ip` (either a domain name or a numeric ip) on its port `port`, a number, optionally sends a string to the server, and then closes the connection again. It resembles the UNIX ping command, but works on a low-level network connection and does not use ICMP.

By default, only one connection attempt is conducted before the function returns. You can specify the number of connection attempts by the optional argument `iters`, a positive integer.

The function waits one second before connecting to the server again. You can change this by passing a different number of seconds for the argument `delay`, a positive integer.

If `message` is not given, the function does not send any data to the server. You can change this by passing a string as argument `message`, which might also be the empty string.

By default, the function prints the connection results at the console with each iteration. This can be suppressed by passing any non-null value as argument `noprint`. If you specify a value for `noprint` and if you do not want to send a string to the server, just pass a non-string value as argument `message`.

The following data is printed at the console if `noprint` is void: Date and time, round-trip time for the current connection in seconds, average round-trip time, a Boolean indicating whether the connection was successful (true) or not (false), and the number of the current iteration. Example:

```
> net.smallping('www.anything.foo', 80, 4, 2)
> # four iterations, 2-second delay, no message
2014/01/01 13:54:30   0.296   0.296   true  1
2014/01/01 13:54:32   0.031   0.163   true  2
2014/01/01 13:54:34   0.047   0.125   true  3
2014/01/01 13:54:36   0.047   0.105   true  4
```

The function returns the date and time of the final iteration as a number indicating the number of seconds passed since a given `epoch`, the average round-trip time in seconds as a number, and a Boolean indicating whether the last connection attempt was successful (**true**) or not (**false**). Use **skycrane.todate** to convert the numeric date into a readable format.

The function is written in Agena and included in the lib/net.agn file.

net.survey ([o], [timeout [, mode [, throw]]])

The function looks for activity on all open sockets, or of specific sockets. If you want to scan only specific sockets, pass a sequence `o` of socket handles as the first argument.

The returns are three sequences and a Boolean: the first sequence with descriptors of sockets ready for reading, the second sequence containing all descriptors of sockets ready for writing, and the third sequence with the descriptors of sockets which encountered exceptional conditions. (Exceptional conditions are not failures.) If the Boolean is **true** then input is available, if it is **false** it indicates a timeout.

By default, **net.survey** waits endlessly and only returns if a network action has been detected (so-called `blocking mode`).

If the positive number `timeout` is passed to the function, the functions will always return after `timeout` seconds even if there was no activity. if `timeout` is **infinity**, it waits endlessly for a connection.

If `mode` is the string `'read'`, then the function only scans sockets ready for reading. If `mode` is the string `'write'`, then the function only scans sockets ready for writing. If `mode` is the string `'except'`, then the function only scans sockets where exceptions occurred. In all three cases, the returns are a sequence of the respective sockets handles and the Boolean **true** if input is available, or **false** at timeout.

If `throw` is set to **false**, then the function does not quit with an error in case the socket status could not be determined.

A socket handle returned can be passed to the **net.accept** function so that an incoming connection can be further processed.

The function is a binding to C's `select` function.

See also: **net.accept**, **net.bind**, **net.listen**, **net.receive**.

net.wget (domain, [path [, port]]])

The function downloads an HTML file from a web server.

`domain`, a string, specifies the domain. `path`, also of type string, indicates the absolute path including the HTML file name on the web server. If `port`, a non-negative integer less than 65,535 is given, then the function tries to query this port instead of the standard HTML port 80.

If only `domain` is given, then it may include the absolute path. If you want to download data from a different port than 80, however, you must pass the absolute path as the second argument.

The function uses the HTTP 1.0 protocol along with the GET method.

The function returns the retrieved web page as a string, including its HTTP protocol header.

Examples:

```
> import net

> net.wget('www.lua.org', 'about.html'):
HTTP/1.1 200 OK
Server: Zeus/4.3
...

> net.wget('www.lua.org/about.html'):
```

The function is written in Agena and included in the lib/net.agn file.

13.2 usb - libusb Binding

As a *plus* package, this library is not part of the standard distribution and must be activated with the **import** statement, e.g. `import usb`.

The package provides 1:1 access to libusb functions. Please have a look at the libusb man pages and is available in the Windows version of Agena, only.

The functions provided by this binding are:

13.2.1 CTX Functions

Package function name	Corresponding libusb function
usb.event_handler_active	libusb_event_handler_active
usb.event_handling_ok	libusb_event_handling_ok
usb.get_device_list	libusb_get_device_list
usb.get_next_timeout	libusb_get_next_timeout
usb.get_pollfds	libusb_get_pollfds
usb.handle_events	libusb_handle_events
usb.handle_events_locked	libusb_handle_events_locked
usb.handle_events_timeout	libusb_handle_events_timeout
usb.lock_event_waiters	libusb_lock_event_waiters
usb.lock_events	libusb_lock_events
usb.pollfds_handle_timeouts	libusb_pollfds_handle_timeouts
usb.set_debug	libusb_set_debug
usb.set_pollfd_notifiers	libusb_set_pollfd_notifiers
usb.try_lock_events	libusb_try_lock_events
usb.unlock_event_waiters	libusb_unlock_event_waiters
usb.unlock_events	libusb_unlock_events
usb.wait_for_event	libusb_wait_for_event

13.2.2 DEV Functions

Package function name	Corresponding libusb function
usb.get_active_config_descriptor	libusb_get_active_config_descriptor
usb.get_bus_number	libusb_get_bus_number
usb.get_config_descriptor	libusb_get_config_descriptor
usb.get_config_descriptor_by_value	libusb_get_config_descriptor_by_value
usb.get_device_address	libusb_get_device_address
usb.get_device_descriptor	libusb_get_device_descriptor
usb.get_max_iso_packet_size	libusb_get_max_iso_packet_size
usb.get_max_packet_size	libusb_get_max_packet_size
usb.open	libusb_open

13.2.3 Handles

Package function name	Corresponding libusb function
usb.attach_kernel_driver	libusb_attach_kernel_driver
usb.bulk_transfer	libusb_bulk_transfer
usb.claim_interface	libusb_claim_interface
usb.clear_halt	libusb_clear_halt
usb.close	closehandle
usb.control_transfer	libusb_control_transfer
usb.detach_kernel_driver	libusb_detach_kernel_driver
usb.get_configuration	libusb_get_configuration
usb.get_descriptor	libusb_get_descriptor
usb.get_device	libusb_get_device
usb.get_string_descriptor	libusb_get_string_descriptor
usb.get_string_descriptor_ascii	libusb_get_string_descriptor_ascii
usb.get_string_descriptor_utf8	libusb_get_string_descriptor_utf8
usb.interrupt_transfer	libusb_interrupt_transfer
usb.kernel_driver_active	libusb_kernel_driver_active
usb.release_interface	libusb_release_interface
usb.reset_device	libusb_reset_device
usb.set_configuration	libusb_set_configuration
usb.set_interface_alt_setting	libusb_set_interface_alt_setting

13.2.4 Transfer Functions

Package function name	Corresponding libusb function
usb.cancel_transfer	libusb_cancel_transfer
usb.control_transfer_get_data	libusb_control_transfer_get_data
usb.control_transfer_get_setup	libusb_control_transfer_get_setup
usb.fill_bulk_transfer	libusb_fill_bulk_transfer
usb.fill_control_setup	libusb_fill_control_setup
usb.fill_control_transfer	libusb_fill_control_transfer
usb.fill_interrupt_transfer	libusb_fill_interrupt_transfer
usb.fill_iso_transfer	libusb_fill_iso_transfer
usb.get_iso_packet_buffer	libusb_get_iso_packet_buffer
usb.set_iso_packet_buffer	libusb_set_iso_packet_buffer
usb.set_iso_packet_lengths	libusb_set_iso_packet_lengths
usb.submit_transfer	libusb_submit_transfer
usb.transfer_get_data	libusb_transfer_get_data

13.2.5 Miscellaneous Functions

Package function name	Corresponding libusb function
usb.init	libusb_init
usb.open_device_with_vid_pid	libusb_open_device_with_vid_pid
usb.transfer	libusb_transfer

13.3 com - Serial RS-232 Communication through COM Ports

As a *plus* package, the **com** package is not part of the standard distribution and must be activated with the **import** statement, i.e. `import com`.

The **com** library allows to send and receive data through physical and virtual COM ports. It is currently in an experimental state.

Typical usage for sending data to another com port:

```
> import com

> fd := com.open('COM4');

> com.init(fd, 9600, "sb1"):

> com.attrib(fd):

> com.write(fd, 'hallo !');

> com.close(fd);
```

Reading data from a com port:

```
> fd := com.open('COM4');

> com.init(fd, 9600, "sb1"):

> do                              # if nothing has been read, com.read ...
>   str := com.read(fd, 80);             # returns null ...
>   if str :: string then print(str) fi  # and a string otherwise
> od;
```

The functions included are:

com.open (port)

Creates a handle to the given COM port `str`, a string, and returns the handle, of type userdata.

In Windows, a COM port may be denoted by the strings `'COM1'`, `'COM2'`, etc.

The function does not configure the port, see **com.init**.

See also: **com.close**.

com.close (hdl)

Closes the given port denoted by its handle `hdl` of type userdata.

See also: **com.open**.

`com.write (hdl, str)`

Sends the entire string `str` to the port denoted by `hdl`. The number of bytes actually sent will be returned.

See also: **com.read**.

`com.read (hdl [, bufsize])`

Reads data from the port denoted by `hdl` and returns it as a string. If nothing could be read, the return is **null**.

The (maximum) number of bytes to be read is given by `bufsize`. If not given, the function tries to read the entire in-queue. You should prefer to pass a value for `bufsize`.

See also: **com.write**.

`com.attrib (hdl)`

Returns the settings of the given COM port, denoted by its handle `hdl`. The function is available in Windows only.

`com.init (hdl, option₁, options₂, ···)`

Configures the COM port denoted by its handle `hdl`. The function re-initialises all hardware and control settings, but it does not empty output or input queues.

Supported settings are:

- `'reset'` - reset
- baud rate, a number,
- `'cs5'` to `'cs8'`, - character size,
- `'parno'`, `'parodd'`, `'pareven'` - parity
- `'sb1'`, `'sb2'` - stop bits,
- `'foff'`, `'frtscts'`, `'fxio'` - flow controls.

Example:

```
> com.init(fd, 9600, "sb1");
```

In DOS, you (obviously ?) can only use baud rates 9600, 19200 and 38400. At least in UNIX, supported baud rates are:

9600, 19200, 38400, 57600, 115200, 230400, 460800, 500000, 576000, 921600, 1000000, 1152000, 1500000, 2000000, 2500000, 3000000, 3500000, 4000000

`com.control (hdl, option1, ···)`

Sets DTR/DSR or RTS/CTS hardware flow controls to the COM port denoted by its handle `hdl`. Options are:

- `'dtr'` - Data Terminal Ready
- `'dsr'` - Data Set Ready
- `'rts'` - Request To Send
- `'cts'` - Clear To Send

The function does not empty output or input queues.

`com.timeout (hdl, msecs)`

Sets the time-out parameters for all read and write operations on the specified COM port, denoted by its handle `hdl`. `msecs` is in milliseconds.

`com.queues (hdl, in_buffersize, out_buffersize)`

Sets buffer sizes for in and out queues for the COM port denoted by its `hdl` handle. `*buffersize`s are integers.

`com.purge (hdl, mode)`

Discards all characters from the output or input buffer of the given COM port, denoted by it handle `hdl`. It can also terminate pending read or write operations on the resource.

`mode` may be the string: `'rw'`, `'r'` or `'w'`.

`com.wait (hdl, event1, ···)`

Waits for an event to occur for the COM port denoted by its handle `hdl`. The set of events that are monitored by this function are the strings: `'car'`, `'cts'`, `'dsr'`, `'ring'`.

The function returns a Boolean.

Chapter **Fourteen**

System & Environment

14 System & Environment

14.1 os - Access to the Operating System

This library is implemented through table `os`.

To determine the operating system and CPU in use by Agena, see the **environ.os** and **environ.cpu** environment variables explained in Appendix A3.

Summary of functions:

File and directory handling:

> **os.chdir, os.chmod, os.chown, os.curdrive, os.curdir, os.dirname, os.exists, os.fattrib, os.fcopy, os.filename, os.fstat, os.ftok, os.gettemppath, os.inode, os.isdir, os.isfile, os.islink, os.iterate, os.list, os.listcore, os.mkdir, os.move, os.prefix, os.readlink, os.realpath, os.remove, os.rmdir, os.suffix, os.symlink, os.tmpdir, os.tmpname, os.whereis.**

Hardware access:

> **os.battery, os.beep, os.cdrom, os.endian, os.freemem, os.hasnetwork, os.isdocked, os.ismounted, os.isremovable, os.isvaliddrive, os.meminfo, os.screensize.**

Operating System Access:

> **os.codepage, os.computername, os.cpuinfo, os.cpuload, os.drives, os.drivestat, os.environ, os.execute, os.exit, os.getenv, os.getlanguage, os.getlocale, os.getmodulefilename, os.groupinfo, os.isansi, os.isarm, os.isdos, os.islinux, os.islocale, os.isos2, os.isppc, os.isunix, os.iswindows, os.isx86, os.login, os.netdomain, os.netsend, os.netuse, os.os2info, os.pause, os.pid, os.setenv, os.settime, os.setlocale, os.system, os.terminate, os.userinfo, os.wait, os.winver.**

Date and Time:

> **os.clock, os.date, os.datetosecs, os.difftime, os.esd, os.isdst, os.lsd, os.now, os.secstodate, os.speed, os.time, os.tzdiff, os.uptime, os.usd.**

`os.battery ()`

On Windows 2000 and later, the function returns the current battery status of your system (usually laptops) as a table with the following information:

Key	Meaning
`'acline'`	'on', 'off', or 'unknown'
`'installed'`	**true** if a battery is present, and **false** otherwise
`'life'`	battery life in percent; a value > 100 indicates that a battery is not installed (see `'status'` entry)
`'status'`	either 'low' (capacity < 33%), 'medium' (capacity > 32% and <67 %), 'high' (capacity > 66%), 'critical' (capacity < 5%), 'charging', 'no battery', 'unknown'
`'charging'`	**true** if battery is currently being charged, or **false** otherwise
`'flag'`	the battery flag, a number
`'lifetime'`	the remaining battery lifetime in seconds, a number (or **undefined** if it could not be determined)
`'fulllifetime'`	the battery lifetime in seconds when at full charge, a number (or **undefined** if it could not be determined)

On OS/2 Warp 4 and higher, with APM running, the functions returns the status of the battery as a table with the following information:

Key	Meaning
`'acline'`	'on', 'off', 'unknown', or 'invalid'
`'life'`	battery life in percent, or 'undefined' if not available
`'status'`	either 'high', 'low', 'critical', 'charging', 'unknown', or 'invalid'
`'flags'`	OS/2 power flags
`'power-management'`	**true** if power management is switched on, or **false** if not.

On other operating systems, the function returns **fail**.

`os.beep ()`

`os.beep (freq, dur)`

In the first form, the functions sounds the loudspeaker with a short `beep` and returns **null**.

The second form sounds the loudspeaker with frequency `freq` Hz (a positive integer in the range 37 .. 32767) for `dur` seconds (a positive float) in Windows, DOS and OS/2. In UNIX, the loudspeaker beeps `dur` times, and the frequency is ignored (just pass any number to `freq`). Returns **null** if a sound could be created successfully, or **fail** if non-positive arguments were passed.

`os.cdrom (d, action)`

Opens and closes the tray of an optical disk drive `d`. It can also eject any other removable drive d. If `action` is `'open'` or `'eject'`, the tray is opened or the media is ejected. If `action` is `'close'`, the tray is closed. The function is available in the OS/2, Linux, and Windows edition of Agena only.

See also: **os.unmount**.

`os.chdir ([str [, any]])`

Changes into the directory given by string `str` on the file system. Returns **true** on success and issues an error on failure otherwise if any has not been given. If you pass any second argument, the function will not issue an error and will return **false** if the given path does not exist, or **fail** if you have no permission to enter the directory.

If no argument is given or **null** is passed for `str`, the name of the current working directory will be returned as a string. Otherwise, the function will commit the change of directory and return **true**.

See also: **os.curdir**, **os.dirname**, **os.isdir**, **os.iterate**, **os.list**.

`os.chmod (fn, m)`

Takes a file path `fn` (a filename, thus a string) and a mode `m` (an integer) denoting a three-digit octal number and changes the file permissions accordingly. Contrary to **os.fattrib**, mode `m` must *not* be preceded by the `` `0o` `` token. The function returns **true** on success and issues an error otherwise. It is available in the UNIX versions of Agena, only.

See also: **os.chown**, **os.fattrib**.

`os.chown (fn, o [, g])`

The function changes the owner of the file `fn` (a filename, thus a string) to owner `o`, and optionally to group `g`.

`o` and `g` may be numbers or strings. If a number is passed for `o` or `g`, it denotes a user id (uid) or group id (gid), respectively. If a string is passed, it denotes a user or group name. If `g` is not given, the default group of user `o` is set.

The function returns true on success and issues an error message otherwise. It is available in the UNIX versions of Agena, only.

See also: **os.chmod**, **os.fattrib**.

`os.clock ()`

Returns the processor time used by Agena. Divide the return by **environ**.**kernel**().**clockspersecond** to get the number of seconds.

See also: **time**, **os.time**.

`os.codepage ()`

`os.codepage (p)`

In Windows, in the first form, returns - in the following order: input code page, output code page, the input code page name and the output code page name.

In OS/2, in the first form, returns - in the following order: input code page and output code page. In DOS, in the first form, returns the output code page.

In OS/2 and Windows only, in the second form, sets the input and output code page. To change the input code page, pass the pair 'input':<code page number>. To change the output code page, pass the pair 'ouput':<code page number>.

In all other operating systems, the function is not available.

See also: **os.getlocale**, **os.setlocale**.

`os.computername ([option])`

Returns the name of the computer in Windows, OS/2, DOS, Mac OS X and UNIX. The return is a string. On other architectures, the function returns **fail**.

If called with any `option`, returns detailed information on the NetBIOS or DNS name associated with the local computer in a table.

See also: **os.netdomain**.

`os.cpuinfo ()`

Returns various information on the CPU in use: its type, frequency, and number of cores. It is available in Windows 2000 and later, OS/2, DOS, Solaris, Linux, and Mac OS X only[26]. The return is a table with the following fields:

---

[26] In Solaris, you may issue `io.pcall('kstat')` and parse its return.

Field	Meaning	OS/2	Win-dows	Mac	Linux
`'bigendian'`	endianness: **true** means Big Endian, **false** Little Endian, and **fail** undetermined.	x	x	x	x
`'bogomips'`	crude measurement of CPU speed				x
`'brand'`	processor name, a string[27]		x	x	x
`'cpuid'`	detailed information on the underlying CPU hardware returned by the C function __cpuid.		x		
`'cputype'`	detailed information on the CPU		x		
`'frequency'`	clock rate in MHz, a posint		x	x	x
`'level'`	processor level, a posint		x	x	
`'model'`	processor model, a posint			x	x
`'ncpu'`	number of cores, a posint	x	x	x	
`'revision'`	processor revision, a posint		x		
`'stepping'`	processor stepping, a posint			x	x
`'support'`	supported instruction sets		x		
`'type'`	architecture: in Windows the string: 'x86', 'x64', 'ARM', 'Itanium', or 'unknown'; on a Mac: 'x86', 'x64', 'ppc', 'ppc64', 'MC680x0', 'MC88000', MC98000', HPPA', 'ARM', 'sparc', 'i860', or 'unknown'. In Linux: a posint.	x	x	x	x
`'vendor'`	vendor ID, e.g. 'GenuineAMD', 'GenuineIntel'.		x	x	x

On all supported operating systems, all data is determined by querying the first processor on the platform, assuming that all other cores have the same features. The returns may be platform-dependent - especially, the return regarding 'level' may have a different meaning.

On other platforms, the function returns **fail**.

The Linux version has been written in Agena, see the lib/library.agn file; the other OS versions have been implemented in C.

See also: **os.cpuload**, **os.endian**.

`os.cpuload ()`

In OS/2, Linux and Mac OS X, returns the 1, 5 and 15 minute load averages of the computer as a sequence of three numbers in the range [0 , 1]. In Windows Vista and later, it just returns a sequence containing the current average load, the load caused by the kernel and the load caused by user programmes - all three in the range [0, 1] - plus the number of elements in the CPU queue, the number of

---
[27] The return may include leading or trailing blanks.

context switches per second and the number of interrupts per second - in this order.

If the load could not be returned, the function just returns **fail**.

On all other platforms, the function returns **fail**.

See also: **os.cpuinfo**, **os.speed**.

os.curdir ()

Determines the current working directory and returns its absolute path.

See also: **os.chdir**.

os.curdrive ()

In OS/2, DOS, and Windows returns the letter of the current drive, a one-character string with an appending colon.

os.date ([format [, time]])

os.date ([format [, obj]])

os.date (format, year, month, day [, hour [, minute [, second]]])

Returns a string or a table containing date and time, formatted according to the given string `format`.

If the `time` argument is present, i.e. the number of seconds elapsed since a given epoch (usually January 01, 1970, or try `os.now(0)`), this is the time to be formatted. Otherwise, **date** formats the current time. To convert a date and time to seconds, see **os.datetosecs**.

In the second form, receives a `format` and a date and optionally time of the form `year`, `month`, `date` [, `hour` [, `minute` [, `second`]]], with all values in table, sequence or register `obj`. Alternatively in the third form, `year`. `month`, `day` and optionally `hour`, `minute`, and `second` can be passed directly.

If `format` starts with '!', then the date is formatted in Co-ordinated Universal Time. After this optional character, if `format` is *t, then **date** returns a table with the following fields: `year` (four digits), `month` (1..12), `day` (1..31), `hour` (0..23), `min` (0..59), `sec` (0..59), `msec` (0..999) - if milliseconds could be determined, `wday` (weekday, Monday is 1, Sunday is 7), `yday` (day of the year, where 1 is January 01, and December 31 either 365 or 366), and `isdst` (daylight saving flag, a boolean). By setting environ.kernel(iso8601 = false), the weekday return 1 means Sunday and 7 Saturday.

If the format is *j, the Julian date, a number, will be returned. If the format is *l, the Lotus 1-2-3 Serial Date, a number, will be returned. For more information on the Lotus Serial Date value returned, see **os.lsd**. If the format is *e, the Excel Serial Date,

a number, will be returned, see **os.esd**. `*sdn` computes the Julian date in the Julian calendar (whereas `*j`, `*l`, `*e` , `*t` compute it in the Gregorian calendar).

If `format` is not `*t`, `*e`, `*l`, `*j`, or `*sdn`, then **date** returns the date as a string, formatted according to the same rules as the C function `strftime`.

When called without arguments, **os.date** on all supported platforms returns a string of the format 'YYYY/MM/DD mm:hh:ss.xxx', where .xxx denotes milliseconds, if they could be determined; otherwise the return would simply be in the format "YYYY/MM/DD mm:hh:ss".

Examples:

```
> os.date('%a, %d %b %Y %H:%M:%S, %z'):
Mon, 02 Nov 2015 17:22:09, W. Europe Standard Time

> os.date('%A, %d %B %Y %H:%M:%S, %z'):
Monday, 02 November 2015 01:02:28, W. Europe Standard Time
```

The following date specifiers always refer to the current locale, and may not be fully supported by your operating system, if not, an empty string will be returned:

Spec	Meaning	Example (w/o the quotes)
%a	abbreviated weekday name	'Fri' for Friday
%A	full weekday name	'Friday'
%b	abbreviated month name	'Apr' for April
%B	full month name	'April'
%c	preferred calendar time representation	'05/19/17 20:33:13' for UK
%C	century of the year, greatest integer not greater than year divided by 100.	does not work on Windows
%d	day of the month as two-digit integer	'03'
%D	date of the format %m/%d/%y	does not work on Windows
%e	day of the month like with %d, but padded with blank instead of a zero	does not work on Windows
%F	date of the format %Y-%m-%d	does not work on Windows
%g, %G	year corresponding to the ISO week number, but without the century	do not work on Windows
%h	abbreviated name of month	does not work on Windows
%H	two-digit hour in the range 00 .. 23	'01'
%I	two-digit hour in the range 00 .. 12	'01'
%j	three-digit day of year in the range 001 .. 366	'001'
%k	hour in the range 0 .. 23, like %H, padded with a blank	does not work on Windows
%l	hour in the range 0 .. 12, like %H, padded with a blank	does not work on Windows
%m	month in the range 01 .. 12	'05'
%M	minute in the range 00 .. 59	'01'
%n	newline	does not work on Windows

Spec	Meaning	Example (w/o the quotes)
%p	AM or PM	'PM'
%P	am or pm	does not work on Windows
%R	hour and minute, like%H:%M	does not work on Windows
%s	number of seconds since the epoch	does not work on Windows
%S	seconds in the range 00 .. 60	'01
%T	time of day of format %H:%M:%S	does not work on Windows
%u	day of week as a decimal number range 1 (Monday) to through 7	does not work on Windows
%U, %V	week number in the range 00 ..53, starting with the first Sunday as the first day of the first week	'18'
%w	day of week in the range 0 (Sunday) to 6	'1'
%W	week number of current year in the range 00 .. 53, starting with the first Monday as the first day of the first week	'18'
%x	preferred date representation	'05/03/17' for UK
%X	preferred time of day representation	'21:09:02' for UK
%y	year in the range 00 .. 99	'17'
%Y	full year number	'1949'
%z	numeric time zone	Windows returns time zone as words
%Z	abbreviation of time zone	Windows returns time zone as words
%%	character '%'	'%'

Please note that the behaviour is undefined if the date passed is earlier than the epoch.

See also: **astro.cweek**, **astro.hdate**, **os.now**, **os.time**, **utils.checkdate**.


```
os.datetosecs ([obj])
```

```
os.datetosecs (year, month, day [, hour [, minute [, second]]])
```

In the first form, receives a date and optionally time of the form `year`, `month`, `date` [, `hour` [, `minute` [, `second`]]], with all values in table, sequence or register `obj` being integers, and transforms it to the number of seconds elapsed since the start of an `epoch` (usually January 01, 1970, try `os.now(0)`). By default, `hour`, `minute`, and `second` are 0. and If no argument is given, returns the number of seconds elapsed from the epoch till the current date and time.

In the second form, receives the given integers, and conducts the same operation.

The time zone acknowledged may depend on your operating system.

The function returns -1 if the date is older than the start of the epoch.

See also: **os.time**, **os.secstodate**, **utils.checkdate**.

os.difftime (t2, t1)

Returns the number of seconds from time `t1` to time `t2`. In POSIX, Windows, and some other systems, this value is exactly `t2-t1`.

See also: **time**, **os.time**.

os.dirname (path)

Returns the directory name of the given `path`, a string. If `path` has no separator, then the function returns `'.'`. If you would like to test relative paths, apply **os.realpath** to `path` before calling this function.

See also: **os.filename**, **os.isdir**, **os.prefix**, **os.suffix**.

os.drives ()

In Windows, OS/2 and DOS, the function returns all the logical drives available at the local computer. The return is a sequence of drive letters. In DOS, floppy drives are not checked to avoid "Insert floppy disk" messages. In other systems, the return is **fail**.

os.drivestat (drive)

In Sun Solaris, Linux, OS/2, DOS and Windows, the function returns information of the given logical drive in a table. In DOS-based systems, pass the drive letter followed by a colon. In UNIX, pass just `'.'`. The following data will be returned:

Key	Meaning
`'label'`	the drive label
`'filesystem'`	the file system (e.g. NTFS, FAT32, JFS, HPFS, etc.)
`'drivetype'`	the type of the drive, i.e. 'Removable', 'Fixed', 'Remote', 'CD-ROM', or 'RAMDISK'
`'freesize'`	the number of free space in bytes
`'totalsize'`	the total number of physical bytes
`'totalclusters'`	total number of clusters (OS/2: allocation units)
`'freeclusters'`	number of free clusters (OS/2: allocation units)
`'freeuserclusters'`	number of free clusters to non-superusers (UNIX only)
`'sectorspercluster'`	number of sectors per cluster
`'bytespersector'`	number of bytes per sector
`'maxnamelength'`	maximum number of characters in a filename (Linux only)
`'totalnodes'`	total number of nodes (UNIX only)
`'freenodes'`	number of free file nodes (UNIX only)
`'trim'`	solid-state disk indicator (Windows only)
`'dosdevice'`	DOS device name (Windows only)
`'fatsize'`	number of bits used by the FAT (12, 16 or 32, DOS only)

Key	Meaning
`'isfat32'`	FAT32-formatted drive (DOS only)
`'driveletter'`	drive letter of device, including colon (OS/2 only)
`'iType'`	unknown meaning, PFSQBUFFER2 data type item `iType` (OS/2 only)
`'idFileSystem'`	unknown meaning, FSALLOCATE data type item `idFileSystem`(OS/2 only)

In other systems, the return is **fail**.

Example:

```
> os.drivestat('c:'):  # get information on drive C:\
[bytespersector ~ 512, drivetype ~ Fixed, filesystem ~ NTFS, freeclusters ~
62051077, freesize ~ 254161211392, label ~ <none>, sectorspercluster ~ 8,
totalclusters ~ 122070527, totalsize ~ 500000878592]
```

See also: **os.ismounted**, **os.isremovable**.

os.endian ()

Determines the endianness of your system. Returns 0 for Little Endian, 1 for Big Endian, and **fail** if the endianness could not be determined.

See also: **os.cpuinfo**.

os.environ ()

Returns all environment variables of the underlying operating system and their current settings as a table of key ~ value pairs of type string.

See also: **os.getenv**, **os.setenv**.

os.esd ([year, month, day [, hour [, minute [, second]]]])

os.esd (x)

The function computes the Excel Serial date for the given date or - if no argument is given - the current date and time. The Excel Serial represents the number of days that have elapsed since 31st December 1899, 00:00h, where midnight January 01, 1900 is day 1.

If no argument is given, the current Lotus Serial Date is computed. Otherwise, at least `year`, `month`, and `day` - all numbers - must be given. Optionally, you may add an `hour`, `minute`, or `second`, where all three default to 0. The arguments can also be passed in a table or sequence.

The returns is a number, where the fractional portion represents the decimal time.

In the second form, if the Excel Serial Date x - a number - is given, the function returns the corresponding Gregorian year, month, day, the decimal fraction of the day - in the range [0, 1) -, the hour, minute, and second, all numbers. x may be 60, returning February 29, 1900.

The function implemented here takes no account of daylight saving time (which **os.lsd** does): at Winter time change, it returns the same values for (as an example) 02:00 a.m. before and after time change. Also, there is a `gap` in the values returned at Summer time change between 02:00 a.m. and 03:00 a.m.

In case of a non-existing date or if the date is older than the start of the epoch, the function issues an error. Thus, the function never returns 60 for February 29, 1900, the bug in the original Lotus 1-2-3 formula.

See also: **os.lsd**, **os.now**, **os.usd**, **utils.checkdate**.

os.execute ([command [, option]])

This function is equivalent to the C function `system`. It passes the string `command` to be executed by an operating system shell. It returns a status code, which is system-dependent. If `command` is absent, then it returns non-zero if a shell is available and zero otherwise.

If any `option` is given, the function runs `command` and returns the entire output of the command as one string. Any carriage returns (`'\r'`) are removed from the result, but keeps newlines (`'\n'`) untouched.

See also: **io.pcall**.

os.exists (filename)

Checks whether the given file or directory (`filename` is of type string) exists and the user has at least read permissions for it. It returns **true** or **false**.

See also: **os.exists**.

os.exit ([code])

os.exit (code [, false])

In the first form, calls the C function `exit`, with an optional `code` to be passed to the environment in which Agena has been started, to terminate the host programme. The default value for `code` is the success code, usually 0. (In Windows, query ERRORCODE in the shell for the exit status.)

The function by default also closes the interpreter state - this can be prevented by passing the optional Boolean value **false**.

```
os.faccess (path [, mode])
```

```
os.faccess (path [, flags])
```

Checks whether a directory or file can be accessed. `path` represents the path to the file, directory or symbolic link.

In the first form, `mode` specifies the accessibility through the integer constants:

- **os.f_ok** = check for existence
- **os.x_ok** = file is executable
- **os.w_ok** = write access is granted
- **os.r_ok** = read access is permitted (the default)

You can either specify the constant **os.f_ok** for `mode`, or a bitwise-OR mask of the constants **os.r_ok**, **os.w_ok** and/or **os.x_ok** created by calling the **||** operator, see example below.

Please note that the integer values these constants represent vary across platforms, so it is recommended to always use the constant names in the calls to the function.

In the second form, a string `flags` of one or more unique characters, in any order, specifies the permissions to be queried:

- `'f'` = check for existence
- `'x'` = file is executable
- `'w'` = write access is granted
- `'r'` = read access is permitted (the default)

You cannot mix the `'f'` flag with the other flags.

In both forms, if at least one bit in the mask asked for a permission is denied, the function returns **false**, and **true** otherwise. If the specified access is not granted, a string describing the kind of error will be returned, too.

Example:

```
> # check for read access to a non-existent folder
> os.faccess('nofolder'):
false    file or directory does not exist

> # check for both read and write access to an existing file
> os.faccess('myfile.txt', os.r_ok || os.w_ok):
true

> # dito
> os.faccess('myfile.txt', 'rw'):
true
```

See also: **os.fattrib**, **os.fstat**, **os.exists**.

```
os.fattrib (fn, mode)
os.fattrib (fn, oct)
os.fattrib (fn, time)
```

In the first form, sets or deletes file permission flags given by the `mode` string to the file denoted by the filename `fn`.

The `mode` argument must consist of at least three characters and have the following form:

Character 1	Character 2	Character 3, etc.
`'u'` - user	`'+'` - add permission	`'r'` - read permission
`'g'`- group	`'-'` - remove permission	`'w'` - write permission
`'o'`- other		`'x'` - execute permission
`'a'`- user, group, and other		

The first character in `mode` denotes the owner of the file, the second character indicates whether to set or delete a permission, and the following characters indicate which permissions to set or remove.

In Windows and OS/2 the following permission flags are additionally supported:

Character 3, etc.
`'a'` - archive flag
`'s'` - system flag
`'h'` - hidden flag
`'r'` - read-only flag

In the second form, the file mode is set according to the *octal* number `oct`. This number is the same as the numeric argument to the UNIX chmod command, so - for example - pass 0o444 (instead of 444) to the function to set a file to read-only mode for all users.

In the third form, the function changes the modification and access time of the file denoted by its name `fn` to the date and time given in table `time`. The table must include at least integers representing a year, month, and day. It may optionally include an hour, a minute, and a second. If they are missing, they default to zero.

File timestamps can only be changed in UNIX, Windows, Mac OS X , and DOS.

The function returns **true** on success, and **fail** otherwise.

Examples:

```
> os.fattrib('file.txt', 'a-wx');  # deletes write and execute permissions

> os.fattrib('file.txt', 0o444);   # sets read-only for all users
```

```
> os.fattrib('file.txt', [2012, 05, 23, 12, 30, 0]); # sets timestamp
```

See also: **os.fstat**, **os.now**.

os.fcopy (infile, outfile [, overwrite])

os.fcopy (infile, dir [, overwrite])

In the first form, copies the file and its permissions denoted by the filename `infile` to the new file called `outfile`. If `outfile` already exists, an error will be issued, but you may overrule this by passing **true** for `overwrite`. The function internally uses **environ.kernel['buffersize']** for the number of bytes to be copied at the same time, which you may change to another positive integer.

In the second form, the function copies the file `infile` to the existing directory `dir`.

The function returns **true** on success, and **fail** and `infile` otherwise. It also returns **fail** and `infile` if the file could be copied, but the file permissions could not be set. The function issues errors if a file could not be read or created, or if the source and target file are identical.

Use **skycrane.fcopy** if you want to use wildcards/file globbing.

os.filename (path [, option])

Returns the filename of the given `path`, a string. This is equivalent to the C basename function. If you would like to test relative paths, apply **os.realpath** to `path` before calling this function.

If any `option` is given in Windows, then the 8.3 DOS filename is also returned if the file exists, otherwise **null** will be returned as the second result.

See also: **os.dirname**, **os.isfile**, **os.prefix**, **os.suffix**.

os.freemem ([unit])

Returns the amount of free physical RAM available on Windows and Mac OS X and UNIX machines. In OS/2, the function returns the amount of free virtual RAM.

If no argument is given, the return is in bytes. If `unit` is the string `'kbytes'`, the return is in kBytes; if `unit` is `'mbytes'`, the return is in Mbytes; if `unit` is `'gbytes'`, the return is in Gigabytes; if `unit` is `'tbytes'`, the return is in TeraBytes. On other architectures, the function returns **fail**.

See also: **environ.used**, **os.meminfo.**

os.fstat (fn)

Returns information on the file, symbolic link (UNIX and Windows only), or directory given by the string `fn` in a table.

The table includes the following information:

Key	Meaning
`'mode'`	`'FILE'` if `fn` is a regular file, `'LINK'` if `fn` is a symbolic link (UNIX and Windows only), `'DIR'` if `fn` is a directory, `'CHARSPECFILE'` if `fn` is a character special file (a device like a terminal), `'BLOCKSPECFILE'` if `fn` is a block special file (a device like a disk), or `'OTHER'` otherwise
`'length'`	the size of the file in bytes
`'compressed'`	the compressed size of the file in bytes (Windows only)
`'date'`	last modification date in the form yyyy, mm, dd, hh, mm, ss
`'lastaccess'`	last file access date in the form yyyy, mm, dd, hh, mm, ss
`'attribchange'`	last file attribute change date in the form yyyy, mm, dd, hh, mm, ss
`'perms'`	file attributes coded in a decimal integer, use **math.convertbase** to convert the integer x into its octal representation (from base 10 to base 8).
`'bits'`	The permission bits, a string similar to that in UNIX and DOS, e.g. '-rw-rw-r--:-----' or '----------:-drhas' where the bits to the left of the colon are set in the UNIX and DOS versions of Agena, while in Windows and OS/2, the bits to the right of the colon are set. The letters indicate: 'r' - read permission granted (UNIX & DOS) 'w' - write permission granted (UNIX & DOS) 'x' - execute permission granted (UNIX & DOS) 'd' - indicates directory (OS/2 only) 'r' - readonly file (OS/2 and Windows) 'h' - hidden file (OS/2 and Windows) 'a' - archived file (OS/2 and Windows) 's' - system file (OS/2 and Windows)
`'owner'`, `'group'`, `'other'`	Access permissions to the file or directory are returned with the `owner`, `group` (UNIX only), and `other` (UNIX only) keys which each reference tables with information on `read`, `write`, and `execute` permissions. These tables have the following form: `['read' ~ <boolean>, 'write' ~ <boolean>, 'execute' ~ <boolean>]`, where <boolean> is either **true** or **false**. In OS/2 and Windows, the file attributes `'hidden'`, `'readonly'`, `'archived'`, and `'system'` are also returned in the subtable with key 'owner'.
`'blocks'`	(UNIX only) Disk space occupied by the file, measured in units of 512-byte blocks.

Key	Meaning
`'blocksize'`	(UNIX only) Optimal block size for reading or writing this file, in bytes.
`'device'`	(Windows & UNIX-like OSs only) Volume serial number
`'inode'`	(OS/2, Windows & UNIX-like OSs only) Unique file serial number/index node. In OS/2 & Windows, a *pair* of two 32-bit integers representing the higher and lower parts is returned. On all other systems it is a pair, where the left-hand side is always 0, and the right-hand side a 32-bit integer depicting the actual inode.
`'dosname'`	(Windows only) 8.3 DOS name of the file
`'binarytype'`	binary type of an executable: "Win16", "Win32", "Win64", "DOS", "OS2/16", "PIF", "POSIX" or "unknown"
`'uid'`	(UNIX only) user ID
`'gid'`	(UNIX only) group ID

See also: **os.fattrib**.

os.ftok (path, id)

The function uses the identity of the file, i.e. its index node (inode, see **os.fstat**) named by the given `path` name, a string, and the least significant 8 bits of `id` (which must be non-zero) to generate a signed 4-byte integral System V IPC (Inter Process Communications) key.

On UNIX platforms, the return is one integer, and on OS/2 & Windows two integers, the first for the higher part of the file index and the second for its lower part. On all other systems, the function issues an error.

The result is the same for all `path`names that name the same file, when the same value of `id` is used.

The value returned should be different when the (simultaneously existing) files or the given `id`s differ.

See also: **hashes.ftok**.

os.getenv (varname)

Returns the value of the system environment variable `varname`, or **null** if the variable is not defined.

See also: **os.setenv**, **os.environ**.

`os.getextlibpath ()`

In OS/2, returns the current paths to be searched before and after system LIBPATH, when trying to locate DLLs.

The first return is the path to be searched before the LIBPATH, the second one after the LIBPATH.

The function is not available in other operating systems.

See also: **os.setextlibpath**.

`os.getlanguage (id)`

Returns the name of the language by number `id`. The return is a string. Available in Windows only. See also: **os.getlocale**.

`os.getlocale ()`

Returns various information on the current locale including decimal point and thousands separators, currency, and monetary formatting suggestions. The return is a table of the key~value pairs listed below. A value of '' (the empty string) means `unspecified`.

Key	Value	Value type
`'locale'`	current locale of your system	string
`'charset'`	current character classification category of the C locale	string
`'decimal_point'`	decimal-point separator	string
`'thousands_sep'`	thousands separator	string
`'int_curr_symbol'`	international currency symbol according to international standard ISO 4217 "Codes for the Representation of Currency and Funds"	string
`'currency_symbol'`	local currency symbol like XXX	string
`'mon_decimal_point'`	decimal point separator for monetary amounts	string
`'mon_thousands_sep'`	thousands separator for monetary amounts	string
`'positive_sign'`	symbol for positive values	string
`'negative_sign'`	symbol for negative values	string
`'int_frac_digits'`	recommended number of decimal places of monetary amounts according to international standard	number
`'frac_digits'`	recommended number of decimal places of monetary amounts according to local standard	number
`'p_cs_precedes'`	recommendation whether currency symbol precedes positive monetary amount	boolean

Key	Value	Value type
`'n_cs_precedes'`	recommendation whether currency symbol precedes negative monetary amount	boolean
`'p_sep_by_space'`	recommendation whether currency symbol and non-negative monetary amount are separated by a blank	boolean
`'n_sep_by_space'`	recommendation whether currency symbol and negative monetary amount are separated by a blank	boolean
`'p_sign_posn'`, `'n_sign_posn'`	indicator how to position the sign for non-negative and negative monetary quantities: 0: currency symbol and quantity to be enclosed in parentheses 1: print sign before quantity and currency 2: print sign after quantity and currency 3: print sign right before currency symbol 4: print sign right after currency symbol any other number: unspecified	number
`'grouping'`	unknown meaning *)	string
`'mon_grouping'`	ditto	string
`'keyboard'`	(Windows only) for an attached keyboard, the return has the following fields: `'hex'` depicts the language ID as a hexadecimal string; and `'name'` contains the associated language name as a string; `'langid'` is the language ID, an integer. `'primarylangid'` includes the primary language ID as an integer; `'sublangid'` represents the sublanguage ID as an integer; `'UserDefaultLangID'` and `'SystemDefaultUILangID'` are integers. `'UserDefaultLangName'` and `'SystemDefaultUILangName'` are strings and represent the respective Windows settings. Warning: On mixed systems (e.g. default language UK and German keyboard) the information returned may be wrong.	table

*) See description of the C function localeconv on the web.

See also: **environ.decpoint**, **os.codepage**, **os.getlanguage**, **os.setlocale**.

`os.getmodulefilename ()`

In Linux and Windows, returns the absolute path to the currently executing programme as a string. In case of error, returns **null**.

`os.gettemppath ()`

Retrieves the path of the directory designated for temporary files, of type string. Note that on non-Windows systems, the function issues an error if the environment variables TEMP, TMP and TMPDIR are all unassigned.

See also: **os.tmpdir**, **os.tmpname**.

`os.groupname (groupname)`

`os.groupname (gid)`

The function receives a group name (a string) or a group id (an integer) and returns a table with keys `'groupname'` denoting the group name (a string) and `'gid'` denoting the group id (a number). It is available in the UNIX versions of Agena, only.

On all other systems, the function just returns **fail**.

See also: **os.login**, **os.userinfo**.

`os.hasnetwork ()`

The function returns **true** if the system is connected to any network, and **false** otherwise. The function is available in Windows, only. The result is usually **true**. On all other architectures, the function returns **fail**.

`os.inode (path)`

On UNIX-like systems and DOS, for the given `path`, a string, the function returns the device ID and the index node (inode), i.e. two unsigned 4-byte integers.

On OS/2 and Windows, the function returns the volume serial number, an unsigned 4-byte integer, along with two additional unsigned 4-byte integers representing the higher and lower parts of the file index.

On all other platforms, the function issues an error.

`os.isansi ()`

Returns **true** on Agena editions compiled with the LUA_ANSI (strict ANSI C) option, and **false** otherwise.

See also: **os.isdos**, **os.islinux**, **os.isos2**, **os.isppc**, **os.isunix**, **os.iswindows**, **os.isx86**.

`os.isarm ()`

Returns **true** if Agena is run on an ARM CPU, 32- or 64-bit, and **false** otherwise.

See also: **os.isdos**, **os.islinux**, **os.isos2**, **os.isppc**, **os.isunix**, **os.iswindows**, **os.isx86**.

`os.isdos ([any])`

Checks whether Agena is run in the DOS environment and returns **true** or **false**.

The procedure is quite dumb: if you are running the DOS version of Agena, it will always return **true** regardless whether it is actually being run in DOS, OS/2 or Windows.

If you pass `any` argument, the function returns additional information in the following order: the name of the DOS edition, the official major and minor version if available, and the internal major and minor version from which you may deduce the actual DOS version. Furthermore, the internal OEM version number if returned. In FreeDOS, the kernel version number will also be given as the last return.

See also: **environ.os**, **os.isansi**, **os.islinux**, **os.isos2**, **os.isunix**, **os.iswindows**.

`os.isdocked ()`

The function returns **true** if the computer is in docking mode, and **false** otherwise. The function is available in Windows, only. On all other architectures, the function returns **fail**.

`os.isdir (path)`

Checks whether the given `path` refers to a directory and returns **true** or **false**.

See also: **os.chmod**, **os.chown**, **os.dirname**, **os.isfile**, **os.islink**.

`os.isdst ([year, month, day [, hour [, minute [, second]]]])`

Receives a date and optionally time of the form `year`, `month`, `date` [, `hour` [, `minute` [, `second`]]], with all values being integers, checks whether Daylight Saving Time is active for a given date. Alternatively, you may pass the date and time in a table, sequence or register.

The function returns true or false. By default, `hour, minute,` and `second` are 0.

If no argument is given, returns the number of seconds elapsed from the epoch till the current date and time.

The function issues an error if a non-existing date has been passed.

See also: **os.date**, **os.tzdiff**, **utils.checkdate**.

`os.isfile (path)`

Checks whether the given `path` refers to a file and returns **true** or **false**.

See also: **os.chmod**, **os.chown**, **os.filename**.

`os.islink (path)`

Checks whether the given `path` refers to a link and returns **true** or **false**.

See also: **os.chmod**, **os.chown**, **os.filename**.

`os.islinux()`

The function determines whether Agena runs on Linux and returns **true** or **false**.

The function is written in Agena and included in the `libary.agn` file.

See also: **environ.os**, **os.isansi**, **os.isdos**, **os.isos2**, **os.isunix**, **os.iswindows**.

`os.islocale(l)`

The function checks whether the given locale `l` - represented as a string - is supported by the operating system and returns **true** or **false**. If the locale is supported, a description (of type string) will be returned as a second result.

The function - contrary to **os.setlocale** - never changes the current locale.

Examples for locales are `'UK'` for the United Kingdom (at least in Windows localised to the United Kingdom), `'he_IL'` for Hebrew (Israel), `'de_AT'` for Austrian and `'zh_Hans_SG'` for Simplified Chinese (Singapore).

`os.ismounted (d)`

Checks whether the given drive `d` has been mounted. It is available in the Windows edition of Agena only.

See also: **os.cdrom**, **os.drivestat**, **os.isremovable**, **os.isvaliddrive**.

`os.isos2()`

The function determines whether Agena runs on OS/2 and returns **true** or **false**.

See also: **environ.os**, **os.isansi**, **os.isdos**, **os.islinux**, **os.isunix**, **os.iswindows**.

`os.isppc ()`

Returns **true** if Agena is run on a PowerPC CPU, 32- or 64-bit, and **false** otherwise.

See also: **os.isarm**, **os.isdos**, **os.islinux**, **os.isos2**, **os.isunix**, **os.iswindows**, **os.isx86**.

`os.isremovable (d)`

Checks whether the given drive `d` is removable. It is available in the Windows edition of Agena only.

See also: **os.cdrom**, **os.drivestat**, **os.ismounted**, **os.isvaliddrive**.

`os.isunix ()`

Returns **true** if Agena is being run in a UNIX environment (i.e. Solaris, Linux, and OpenSolaris), and **false** otherwise.

The function is written in Agena and included in the `libary.agn` file.

See also: **environ.os**, **os.isansi**, **os.isdos**, **os.islinux**, **os.isos2**, **os.iswindows**.

`os.isvaliddrive (d)`

Checks whether the given drive `d` is part of the file system. It is available in the Windows edition of Agena only.

See also: **os.cdrom**, **os.drivestat**, **os.ismounted**, **os.isremovable**.

`os.iswindows ()`

Checks whether the Agena version for Windows is being run and returns **true** or **false**.

See also: **environ.os**, **os.isansi**, **os.isdos**, **os.islinux**, **os.isos2**, **os.isunix**.

`os.isx86 ()`

Returns **true** if Agena is run on an x86-compatible CPU, 32- or 64-bit, and **false** otherwise.

See also: **os.isarm**, **os.isdos**, **os.islinux**, **os.isos2**, **os.isppc**, **os.isunix**, **os.iswindows**.

`os.iterate (path [, option])`

Creates a factory that when called traverses a directory `path` and `option`ally also returns the type (file, directory, link, etc.). The first argument must be the path to the (sub)directory to be iterated - you can also use the `'.'` , `'*'` or `'*.*'` abbreviations instead. The iterator returns the name of the file, link, directory, etc.

The optional boolean second argument **true** causes the iterator to additionally return the type, i.e. `'FILE'`, `'DIR'`, `'LINK'`, `'CHARSPECFILE'`, `'BLOCKSPECFILE'` or `'OTHER'`. After the directory has been completely traversed, the function returns **null**.

The iterator cannot recurse into a subdirectory, just create another factory instead.

See also: **os.chdir**, **os.list**, **os.listcore**.

`os.list (d [, options])`

Lists the contents of a directory `d` (given as a string) by returning a table of strings denoting the files, subdirectories, and links. The second return is a string with the absolute path to the main directory scanned. If `d` is **null** or the empty string, the current working directory is evaluated. If the return is **null** and a warning text, then `d` does not exist.

`d` may include the `?` and `*` jokers known from UNIX, OS/2, DOS or Windows to select a subset of files, e.g. `os.list('*.c')` to return all files with suffix `.c`. Jokers can only be used to select files, but not to parse subdirectories if they exist.

If no option is given, files, links, and directories are returned. If the optional argument `'files'` or `'file'` is given, only files are returned. If the optional argument `'dirs'` or `'dir'` is given, directories are returned exclusively. If the optional argument `'links'` or `'link'` is given, links are returned (UNIX only). The `'r'` option forces a recursive descent into all subfolders of `d`. Multiple options can be given.

If `d` is `'.'`, then the current working directory is examined. If `d` is `'..'`, then the directory one level higher than the current one is searched.

If the string `'r'` is passed as an option, the function traverses all subfolders in `d`.

The function is written in Agena and included in the lib/library.agn file.

See also: **os.iterate**.

`os.listcore (d)`
`os.listcore (d [, options] [, pattern])`

In the first form, returns a table with all the files, links and directories in the given path `d`. If `d` is void or the string `'.'`, the current working directory is evaluated. It is the core function used by **os.list**.

In the second form, by giving at least one of the options `'files'` or `'file'`, `'dirs'` or `'dir'`, or `'links'` or `'link'`, the file, directory name, or link names are returned, respectively. These three options can be mixed.

Another option may be a `pattern` of type string which can include the wildcards `?` or `*`. If given, the function will only return those entries which match this pattern.

os.login ()

Returns the login name of the current user as a string. The return is a string. In DOS, the function returns **fail**. See also: **os.username**, **os.groupname**.

os.lsd ([year, month, day [, hour [, minute [, second]]]] [, option])

os.lsd (x)

The function computes the Lotus 1-2-3 Serial Date, which is also used in Excel (known there as `Excel Serial Date`). It represents the number of days that have elapsed since 31st December 1899, 00:00h, where midnight January 01, 1900 is day 1.

The function always returns a Standard Time value even if Daylight Saving Time is active for the given date. By passing the `option` **false**, the function takes into account Daylight Saving Time, however.

In the first form, if no argument is given, the current Lotus Serial Date is computed. Otherwise, at least `year`, `month`, and `day` - all numbers - must be given. Optionally, you may add an `hour`, `minute`, or `second`, where all three default to 0.

The first return is a number, where the fractional portion represents the decimal time. Also, the second return **true** or **false** indicates whether Daylight Saving Time has been active for the current or given date (if no `option` has been passed) so that you may add 1/24 to the first result to receive a value Microsoft Excel would return in DST situations. In case of a non-existing date, the function issues an error. Thus, the function never returns 60 for February 29, 1900, the bug in the original Lotus 1-2-3 formula.

In the second form, if the Lotus Serial Date `x` - a number - is given, the function returns the corresponding Gregorian year, month, day, the decimal fraction of the day - in the range [0, 1) -, the hour, minute, and second, all numbers. `x` may be 60, returning February 29, 1900.

To compute the Julian date from the Lotus Serial Date, add 2415018.5.

See also: **os.esd**, **os.now**, **os.usd**, **utils.checkdate**.

os.meminfo ([unit])

os.memstate([unit])

(Windows, UNIX, Mac OS X, DOS and OS/2 only.) Returns a table with information on current memory usage. With no arguments, the return is the respective number of bytes (integers). If unit is the string `'kbytes'`, the return is in kBytes; if unit is `'mbytes'`, the return is in Mbytes; if unit is `'gbytes'`, the return is in Gigabytes, if unit is `'tbytes'`, the return is in Terabytes.

The resulting table will contain the following values, an 'x' indicates which values are returned on your system.

Key	Description	OS/2	Windows	UNIX	Mac
`'freephysical'`	free physical RAM		x	x	x
`'totalphysical'`	installed physical RAM	x	x	x	x
`'freevirtual'`	free virtual memory	x	x		
`'totalvirtual'`	total virtual memory		x		
`'pagesize'`	page size in bytes	x	x	x	x
`'resident'`	occupied resident pages	x			
`'maxprmem'`	maximum number of bytes available for the active process	x			
`'maxshmem'`	maximum number of shareable bytes available	x			
`'active'`	active memory				x
`'freepagefile'`	current committed memory limit for the current process		x		
`'totalpagefile'`	maximum commitable amount of memory for the current process		x		
`'inactive'`	inactive memory				x
`'speculative'`	unknown meaning, see vm_stat.c source code.				x
`'wireddown'`	memory that cannot be paged out				x
`'reactivated'`	memory reactivated				x

On Mac, the function returns Mach virtual memory statistics. Type `man vm_stat` in a shell to get more information on the meaning of the above mentioned Mac-specific values.

The values returned in DOS should be self-explanatory. `'ems_major'` and `'ems_minor'` are memory manager version numbers.

On other architectures, the function returns **fail**.

See also: **environ.used**, **os.freemem**.

os.mkdir (str)

Creates a directory given by string `str` on the file system. Returns **true** on success, and issues an error on failure otherwise.

The function is available on OS/2, DOS, UNIX, Mac OS X, and Windows based systems only.

`os.monitor (action)`

The function switches the monitor on and off (Windows and Linux), and can also put it on stand-by if the monitor supports this feature (Windows only).

Pass the string `'off'` as the only argument to switch off the monitor; pass `'on'` to switch it on, and `'standby'` to put it into stand-by mode. If no argument is given, the Monitor is switched on (which has no effect, if the screen is already active).

On success, the function returns **true**, and **false** and a string containing the error analysis otherwise.

`os.mouse ([mhd])`

In OS/2, DOS and Windows, the function returns various information on the attached mouse by returning a table with the following entries:

Key	Meaning
`'mousebuttons'`	number of mouse buttons; if more than one mouse is attached, the sum of all mouse buttons is computed
`'hmousewheel'`	Windows only: **true** if the mouse features a horizontal mouse wheel, and **false** if not
`'mousewheel'`	Windows only: **true** if the mouse features a vertical mouse wheel, and **false** if not
`'swapbutton'`	Windows only: **true** if the left and right mouse buttons have been swapped
`'speed'`	Windows only: an integer between 1 (slowest) and 20 (fastest)
`'threshold'`	OS/2 & Windows only: the two mouse threshold values, x and y co-ordinates, as a pair of two numbers
`'mickeys'`	OS/2 only: number of mickeys per centimeter; a mickey is the amount that a mouse has to move for it to report that it has moved
`'inmickeys'`	OS/2 only: mouse data in mickeys, not "pels"
`'rowscale'`	OS/2 only: row scaling factor
`'columnscale'`	OS/2 only: column scaling factor

In OS/2 you have to supply a mouse handle `mhd`, see **os.mouseopen**.

On all other platforms, the function returns **fail**.

`os.mouseflush (mhd)`

In OS/2, flushes the queue of the mouse denoted by `mhd` and returns **true** on success and **false** otherwise. The function is not available in all other operating systems. See also: **os.mouseopen**.

`os.mouseopen ()`

In OS/2 only, returns a mouse handle - an integer - that is needed with other os.mouse* functions.

`os.mouseclose (mhd)`

In OS/2 closes the connection to the mouse denoted by `mhd`. See **os.mouseopen**.

`os.mousestate ([mhd, ] [mask [, threshold]])`

In OS/2 and Windows, returns information on the position of the attached mouse and on button clicks plus some operating system-dependent data. In OS/2 you have to supply a mouse handle `mhd`, see **os.mouseopen**.

If the bit-`mask` is 0b0 (equals 0 decimal, the default), the function does not check whether the mouse is being moved at invocation. If `mask` is 0b1 = 1 decimal), movement is being checked. If `mask` is 0b10 (= 2 decimal), the function in Windows tries to transform absolute to window coordinates (default: false, may not work on every Windows version), and if `mask` is 0b11 (=3 decimal) both coordinates are converted and motion is being tracked.

In Windows, the threshold to detect mouse movement is 0.001 seconds. You can choose other values by passing a number to optional `threshold`. In OS/2, the setting is ignored.

The function is not available in all other operating systems.

Key	Meaning
`'row'`	horizontal position of the mouse
`'column'`	vertical position of the mouse
`'button'`	table of three Booleans: the first for the left mouse button, the second for the middle button, and the third for the right button, where **true** depicts `button clicked` and **false** otherwise (experimental in OS/2)
`'motion'`	mouse is being moved (**true**) or not moved (**false**)
`'eventmask'`	OS/2 only: check EDM/2 Website on the MouGetEventMask API function for a description
`'flush'`	OS/2 only: mouse flush is in progress (**true**, **false** otherwise)
`'blockread'`	OS/2 only: block read is in progress (**true**, **false** otherwise)
`'eventqueuebusy withio'`	OS/2 only: event queue is busy with I/O (**true**, **false** otherwise)

`os.move (oldname, newname [, option])`

Renames or moves a file or directory named `oldname` to `newname`. The function returns **true** on success, and issues an error on failure otherwise.

If you pass the `option` **true**, then the function does not issue an error if `oldname` does not exist or if `newfile` already exists. Instead, the function just returns **fail**.

See also: **skycrane.move**.

os.netdomain (servername)

On Windows, returns the domain name and the name of the primary domain controller (PDC). If `servername` is **null** or not given, the local computer is used.

See also: **os.computername**.

os.netsend (server, user, message)

On older Windows flavours, sends a `message` (a string) to a `user` (a string) on a given `server` (a string). If `server` is **null** or not given, the local computer is used.

os.netuse (letter [, path])

On Windows, connects or disconnects a `drive` letter to a network `path`. The drive letter must be followed by a colon. For drive letters already in use, see **os.drives**.

Example:

```
># connect drive with label 'drive_c' on computer TITANIA
># to drive letter Z:
> os.netuse('z:', '\\\\TITANIA\\drive_c');

> os.netuse('z:');  # disconnect
```

The function returns **true** on success and issues an error otherwise.

os.now ([secs])

os.now (year, month, day [, hour [, minute [, second]]])

Returns rather low-level information on the current or given date and time in form of a dictionary.

If no argument is passed, the function returns information on the current date and time. If a non-negative number is given which represents the amount of seconds elapsed since the start of the epoch (try `os.now(0)`), information on this date and time are determined (see **os.datetosecs** to convert a date to seconds).

In the second form, the given date `year`, `month`, `date` and optionally time `hour`, `minute`, `second`, where all the optional values default to 0, is used. Alternatively, you may pass the date and time in a table, sequence or register.

The `gmt` table in the return of the function represents the current date and time in GMT/UTC. The `localtime` table includes the same information for your local time zone.

The `tz` entry represents the difference between your local time zone and GMT in minutes with daylight saving time cancelled out, and *east* of Greenwich. The `td` entry represents the difference between your local time zone and GMT in minutes including daylight saving time, and *east* of Greenwich. `East of Greenwich` means: A positive integer indicates that your computer is located east of Greenwich, a negative value means that you are in a time zone to the west of Greenwich, and 0 means your computer is using GMT. The `jd` entry features the Julian date and time, the `lsd` key represents the Lotus 1-2-3 Serial Date, also known as Excel Serial Date.

The `seconds` entry is the number of seconds elapsed since some given start time (the `epoch`), which on most operating systems is January 01, 1970, 00:00:00. The `mseconds` entry represents milliseconds; it may be missing if milliseconds could not be determined on your platform. The `dst` entry indicates whether daylight saving time is in effect.

The `gmt` and `localtime` entries have the same structure: it is a table of data of the following order: year, month, day, hour, minute, second, number of weekday (where 0 means Sunday, 1 is Monday, and so forth), the number of full days since the beginning of the year (in the range 0:365), whether daylight saving time is in effect at the time given (0: no, 1: yes), the strings 'AM' or 'PM', the month in English (a string), and the weekday in English (a string).

If the date and time could not be determined, **fail**s are returned.

See also: **utils.calendar**, **utils.checkdate**, **os.datetosecs**, **os.lsd**, **os.secstodate**, **os.time, os.tzdiff**.

os.os2info ()

os.os2info (···)

In the first form, returns all 31 OS/2 settings that can be queried via the C API function DosQuerySysInfo, in a table.

In the second form, you can pass any of the following options, to individually query the current settings, either in upper or lower case:

"QSV_MAX_PATH_LENGTH"	"QSV_TOTPHYSMEM"
"QSV_MAX_TEXT_SESSIONS"	"QSV_TOTRESMEM"
"QSV_MAX_PM_SESSIONS"	"QSV_TOTAVAILMEM"
"QSV_MAX_VDM_SESSIONS"	"QSV_MAXPRMEM"
"QSV_BOOT_DRIVE"	"QSV_MAXSHMEM"
"QSV_DYN_PRI_VARIATION"	"QSV_TIMER_INTERVAL"
"QSV_MAX_WAIT"	"QSV_MAX_COMP_LENGTH"

"QSV_MIN_SLICE"	"QSV_FOREGROUND_FS_SESSION"
"QSV_MAX_SLICE"	"QSV_FOREGROUND_PROCESS"
"QSV_PAGE_SIZE"	"QSV_NUMPROCESSORS"
"QSV_VERSION_MAJOR"	"QSV_MAXHPRMEM"
"QSV_VERSION_MINOR"	"QSV_MAXHSHMEM"
"QSV_VERSION_REVISION"	"QSV_MAXPROCESSES"
"QSV_MS_COUNT"	"QSV_VIRTUALADDRESSLIMIT"
"QSV_TIME_LOW"	"QSV_INT10ENABLED"
"QSV_TIME_HIGH"	

See: http://www.edm2.com/os2api/Dos/DosQuerySysInfo.html.

os.pause ([n [, msg]])

Waits for an amount of time or any user input and emulates the ZX Spectrum command PAUSE: if no argument is given or `n` = 0, the function waits forever until the user presses any key.

If `n` is a positive number, then the function waits for `n` seconds - `n` may be a fraction - unless the user presses any key - in the latter case, the function quits waiting immediately and exits. If an optional string `msg` is given, the function prints it on screen - you may terminate the string with a newline ('\n') to force a linefeed. The function returns the number of seconds the function waited.

See also: **os.wait**.

os.pid ()

Returns Agena's process ID as a number.

os.prefix (filename)

Returns `filename`, a string, without suffix if there is one. If `filename` includes a path, it is included in the return.

The function is written in the Agena language and part of the lib/library.agn file.

See also: **os.dirname**, **os.filename**, **os.suffix**, **strings.chomp**.

os.readlink (linkname)

Returns the target of the symbolic link `linkname` as a string. If the link does not exist or if an error occurred, it returns **fail** and optionally a string indicating the type of error.

In Windows, the function only recognises classical Windows shortcut files, it cannot resolve NTFS symbolic links or junctions.

The function is available in UNIX including Mac OS X, and Windows.

See also: **os.symlink**.

os.realpath (pathname [, option])

Converts the `pathname` argument of type string to an absolute pathname, with symbolic links resolved to their actual targets and no . or .. directory entries. The return is a string. This only works satisfactorily if you are in the current working directory of the file to be scrutinised when calling the function.

In Windows, OS/2 and DOS, the function returns the path with slashes instead of backslashes by default. You can override this by passing an optional **false**.

os.remove (filename [, option])

Deletes the file or directory with the given name. Directories must be empty to be removed. Returns **true** on success, and issues an error on failure otherwise.

If you pass the `option` **true**, then the function does not issue an error if the file to be deleted does not exist. Instead, the function returns **fail**.

os.rmdir (dirname [, option])

Deletes a directory denoted by the string `dirname` on the file system. Returns **true** on success, and issues an error on failure otherwise.

If you pass the `option` **true**, then the function does not issue an error if the directory to be deleted does not exist. Instead, the function returns **fail**.

os.screensize ()

In OS/2 and Windows, returns the current horizontal and vertical resolution of the display as a pair of width:height. On all other platforms, the function issues **fail**.

os.secstodate (secs)

Takes the number of seconds `secs` elapsed since the start of an epoch, in your local time zone, and returns a table of integers in the order: year, month, day, hour, minute, second. In case of an error, **fail** will be returned.

See also: **os.datetosec**.

`os.setenv (var, setting)`

Sets the environment variable in the underlying operating system. `var` must be a string. If `setting` is a string or number, the environment variable `var` is set to `setting`. If `var` has already been assigned before, its value is overwritten.

If `setting` is **null**, then the environment variable `var` is deleted (not supported in DOS).

See also: **os.getenv**, **os.environ**.

`os.setextlibpath (path  [, option])`

In OS/2, sets the `path` to be searched before and after system LIBPATH, when trying to locate DLLs. If `option` is 0, then the path is set to the beginning of LIBPATH; if it is non-zero, the path is set to its end.

The function returns **true** on success and **false** otherwise.

The function is not available in other operating systems.

See also: **os.getextlibpath**.

`os.setlocale (locale [, category])`

Sets the current locale of the programme. `locale` is a string specifying a locale, the empty string returns the locale of your operating system (and not the locale you have set in Agena); `category` is an optional string describing which category to change: `'all'`, `'collate'`, `'ctype'`, `'monetary'`, `'numeric'`, or `'time'`; the default category is `'all'`.

The function returns the name of the new locale, or **null** if the request cannot be honoured.

When called with **null** as the first argument or no argument at all, this function only returns the name of the current locale for the given category.

See also: **os.codepage**, **os.islocale**, **skycrane.getlocales**.

`os.settime (secs)`

`os.settime (year, month, day [, hour [, minute [, second]]])`

In the first form, takes the number of seconds `secs` elapsed since the start of an epoch, in your local time zone, and sets the system clock accordingly. In the second form, the given date `year`, `month`, `date` and optionally time `hour`, `minute`, `second`, where all the optional values default to 0, is used. Alternatively, you may pass the date and time in a table, sequence or register.

Agena must be run in root mode in order to change the system time. In case of an error, **fail** will be returned. The function is available only in the Windows, Solaris, OS/2, and Linux versions of Agena.

See also: **os.datetosecs**, **utils.checkdate**.

os.strerror ([n])

Returns the text message for the given integral error code `n`, or the latest error issued by the underlying operating system if no argument is given. The result varies across platforms.

os.suffix (filename)

Returns the last suffix in `filename` (a string) and also the position (an integer) of the last suffix in `filename`.

If there is no suffix in `filename`, the function returns the empty string and 0 (zero).

The function is written in the Agena language and part of the lib/library.agn file.

See also: **os.dirname**, **os.filename**, **os.prefix**, **strings.chomp**.

os.symlink (target, linkname)

In UNIX, the function creates a symbolic link named `linkname` to the file called `target`. In Windows, the function creates a classical regular Windows shortcut file that points to a real file. It does not create NTFS junctions or NTFS symbolic links.

Both arguments must be strings. The function is not available in DOS.

See also: **os.readlink**.

os.system ()

Returns information on the platform on which Agena is running.

Under Windows, it returns a table containing the string 'Windows', the major version (e.g. 'NT 4.0', '2000', etc.) as a string, the Build (*dwBuildNumber*) as a number, the platform ID (*dwPlatformId*) as a number, the major version (*dwMajorVersion*) as a number, the minor version (*dwMinorVersion*) as a number, the product type (*wProductType*) as a number, maintainence information (*szCSDVersion*) usually depicting an installed service pack as a string, and a summary string combining all the previously mentioned data in a human-readable fashion, all in this order. For an alternative, see: **os.winver**.

In UNIX, Mac OS X, OS/2, and DOS, it returns a table of strings with the name of the operating system (e.g. 'SunOS', 'OS/2' or 'MS-DOS'), the release, the version, and the

machine, in this order. Note that Mac OS X is recognised as 'Darwin'. In OS/2, the major and minor revision, along with the revision, are returned as numbers, as well.

In Linux and Windows, the function also checks whether the underlying platform runs in 32 or 64-bit mode, and returns the result with key `'bits'`.

If the function could not determine the platform properly, it returns **fail**.

See also: **environ.os**.

os.terminate (action)

The function halts, reboots, sleeps, or log-offs Windows, OS/2 or Mac OS X. In Windows, it can also lock the current user session or hibernate the system.

The function makes sure that no data loss occurs: if there is any unsaved data, the function does not start termination and just quits.

To put the system into energy-saving sleep mode, pass the string `'sleep'` as the only argument. To hibernate (save the whole system state and then shut off the PC), pass `'hibernate'` (Windows only); to shut down the computer completely, pass `'halt'`; to reboot the system, pass `'reboot'`; to lock the current user session without logging the user off, pass `'lock'` (Windows only); to log-off the open session of the current user, pass `'logoff'`.

The OS/2 version solely supports system halt (`'halt'` argument, without power-off) and reboot.

By default, the function waits for 60 seconds before initiating the termination process. You can change this time-out period to another number of seconds by setting the optional second argument to any non-negative integer.

On all other platforms, the function returns **fail** and does nothing.

os.time ([obj])

Returns the current time when called without arguments, or a value representing the date and time specified by the given table or sequence `obj`.

If a table is given, it must either:

• have fields `year`, `month`, and `day`, and may have fields `hour`, `min`, `sec`, and `isdst`, see example below,
• or at least three integers representing year, month, day, and optionally also hour, minute, second.

If `obj` is a sequence, it must contain a four-digits year, the month, and the day, all integers, in this order. It may additionally include the hour, the minute, and the

second, all integers, too, in this order. The optional seventh entry must either be the Boolean **true** or **false** and indicates whether daylight saving time is in effect (default is **false**). See example below.

The returned value is a number, whose meaning depends on your system. In POSIX, Windows, and some other systems, this number counts the number of seconds since some given start time (the `epoch`). In other systems, the meaning is not specified, and the number returned by **os.time** can be used only as an argument to **os.date** and **os.difftime**.

If the return is **null**, then the given date lies before the start of the epoch (check `os.now(0)`). The function process dates between the start of 1900 and the end of 2099, only.

If a second number is returned, it will denote the millisecond portion of the current time in the range [0, 999].

Examples:

```
> os.time(['year' ~ 2013, 'month' ~ 5, 'day' ~ 23,
>    'hour' ~ 1, 'min' ~ 2, 'sec' ~ 3]):
1369263723      791

> os.time(seq(2013, 5, 23, 1, 2, 3, false)):
1369267323      791
```

See also: **time**, **os.clock**, **os.date**, **os.datetosecs**, **os.difftime**, **os.now**, **utils.checkdate**.

`os.tmpdir ([p])`

Creates a unique temporary directory from pattern `p` which ends in six 'X' characters and returns its name. By default, the function uses the pattern 'agn_XXXXXX' (non-DOS) or 'agXXXXXX' (DOS).

In case of an error, the function returns **null** and an error message. You have to manually remove the directory if it is not needed any longer.

See also: **os.gettemppath**, **os.tmpname**.

`os.tmpname ()`

Returns a string with a file name that can be used for a temporary file or directory. The file must be explicitly opened before its use and explicitly removed when no longer needed. The same applies to directories: you have to manually create and remove it. Depending on the platform, the name might denote an absolute or relative path.

See also: **os.gettemppath**, **io.tmpfile**, **io.mkstemp**, **os.tmpdir**.

```
os.tzdiff ([secs])
```

```
os.tzdiff (year, month, day [, hour [, minute [, second]]])
```

Computes the difference between the system's local time zone and UTC in minutes, taking into account whether Daylight Saving Time is active, plus a Boolean indicating whether Daylight Saving Time is active.

If no argument is passed, the function uses the current date and time. If a non-negative number `secs` is given - representing the amount of seconds elapsed since the start of the epoch -, this date and time is used to compute the result.

In the second form, the given date `year`, `month`, `date` and optionally time `hour`, `minute`, `second`, where all the optional values default to 0, is used. You may also pass these date and time data as a table, sequence or register.

See also: **os.isdst**, **os.now**.


```
os.unmount (fs [, force])
```

The function unmounts the filesystem `fs`, which has to be passed as a string. If the option **true** is given for the second argument `force`, the function forces a disconnection even if the filesystem is in use by another process. The default is **false**. If your system cannot force a \`umount\`, this flag is simply ignored.

The function works only if Agena is run with superuser rights. Depending on the operating system, it may only unmount filesystems that the UNIX kernel directly supports (in Linux, look into /proc/filesystems folder), e.g. ntfs-3g filesystems using the FUSE driver may not be unmounted.

On success, **os.unmount** returns **true**, and **false** plus a string indicating the error reason, otherwise.

See also: **os.cdrom**, **os.execute**.


```
os.uptime ()
```

Returns the number of seconds a system has been running. It is available in OS/2, Windows, Solaris, and Linux. In Windows, there may be an overflow if the system has been up for more than 49.7 days.


```
os.usd ([year, month, day [, hour [, minute [, second]]]])
```

```
os.usd (x)
```

The function computes the UTC Serial date, a number, for the given date or - if no argument is given - the current date and time, where the time zone is assumed to be UTC.

The UTC Serial represents the number of days that have elapsed since 31st December 1899, 00:00h, where midnight January 01, 1900 is day 1.

If no argument is given, the UTC Serial Date for the current date and time is computed. Otherwise, at least `year`, `month`, and `day` - all numbers - must be given. Optionally, you may add an `hour`, `minute`, or `second`, where all three default to 0.

In the second form, if the UTC Serial Date `x` - a number - is given, the function returns the corresponding Gregorian year, month, day, the decimal fraction of the day - in the range [0, 1) -, the hour, minute, and second, all numbers. `x` may be 60, returning February 29, 1900.

Since the date and time is considered to be UTC, the function implemented here takes no account of daylight saving time: at Winter time change, it returns the same values for (as an example) 02:00 a.m. before and after time change.

Also, there is a `gap` in the values returned at Summer time change between 02:00 a.m. and 03:00 a.m.

In case of a non-existing date or if the date is older than the start of the epoch, the function issues an error.

See also: **os.esd**, **os.lsd**, **os.now**, **utils.checkdate**.

os.username ([groupname])

os.username ([uid])

The function receives a user name (a string) or a user id (an integer) and returns a table with the following key ~ value pairs:

Key	Value
`'username'`	user's login name
`'uid'`	user ID number
`'gid'`	user's default group ID number
`'realname'`	user's real name, etc.
`'homedir'`	user's home directory, or initial working directory (undefined means system default)
`'shell'`	user's default shell (undefined means system default)

If no argument is given, data for the current user is being determined.

The function is available in the UNIX versions of Agena, only. On all other systems, the function just returns **fail**.

See also: **os.login**, **os.groupinfo**.

`os.vga ()`

In OS/2, DOS and Windows, the function returns a table with the following information on the display:

Key	Meaning
`'resolution'`	a pair with the horizontal and vertical number of pixels (OS/2 and Windows)
`'dimension'`	a pair with the number of rows and columns
`'depth'`	an integer indicating the colour depth in bits
`'monitors'`	the number of monitors attached to the system (Windows only)
`'vrefresh'`	the vertical refresh rate in Hertz (Windows only)
`'mode'`	screen mode, an integer (unknown meaning, DOS only)
`'colours'`	number of colours (OS/2 only)

See also: **os.monitor**, **os.screensize**.

`os.wait (x)`

Waits for x seconds and returns **null**. x may be an integer or a float. This function does not strain the CPU, but execution cannot be interrupted. The function is available on OS/2, DOS, UNIX, Mac OS X, and Windows based systems only.

On other architectures, the function returns **fail**.

See also: **os.pause**.

`os.whereis (file, dir [, options])`

Searches case-insensitively for the given `file`, link or directory in directory `dir` and returns a table of all the hits.

If `dir` is not given, it defaults to the current working directory. If `option` is `'r'`, the subfolders in `dir` are also scanned, returning the files found with their relative paths. If `option` is `'s'`, the search is case-sensitive. You can mix both options.

`file` may include the wildcards `?` and `*`, where `?` represents exactly one unknown character, and `*` represents zero or more unknown characters.

Example:

```
> os.whereis('agena*', '..', 'r'):
```

See also: **os.chdir**, **os.list**, **strings.glob**.

`os.winver ([argument])`

This function is an alternative to **os.system** and returns the internal Windows release number (a float). If any `argument` is given, it also returns the service pack major (an

integer) and minor version (an integer), whether the operating system is a workstation (**true**) or server (**false**) and the build number (an integer), in this order.

On all other platforms other than Windows, the function returns **undefined**, which, if used in a relation, always evaluates to **false**.

Internal Version Number	Official Release
5.0	Windows 2000
5.1	Windows XP
5.2	Windows XP 64-Bit Edition, Windows Server 2003, Server 2003 R2
6.0	Windows Vista, Windows Server 2008
6.1	Windows Server 2008 R2, Windows 7
6.2	Windows Server 2012, Windows 8
6.3	Windows Server 2012 R2, Windows 8.1
10.0	Windows Server 2016 & 2019, Windows 10

14.2 environ - Access to the Agena Environment

This package comprises functions to access the Agena environment, explore the internals of data, read settings, and set defaults.

`environ.anames ([option])`

Returns all global names that are assigned values in the environment. If called without arguments, all global names are returned. If `option` is given and `option` is a string denoting a basic or user-defined type (e.g. `'boolean'`, `'table'`, etc.), then all variables of that type are returned.

The function is written in Agena and included in the lib/library.agn file.

`environ.arithstate ()`

The function returns information encoded in a bit field on the kind of numerical exception encountered by the **inc**, **dec**, **mul**, **div**, **intdiv** and **mod** operators:

add, **sub**:
0b0000: no exception
0b0010: very large value to be added to (subtracted from) value close to zero
0b0100: very large values to be added (or subtracted)
0b1000: both operands are close to zero:

mul:
0b0000: no exception
0b0010: very large value to be multiplied by value close to zero
0b0100: very large values to be multiplied
0b1000: both operands are close to zero

div, **intdiv**, **mod**:
0b0000: no exception
0b0001: denominator is zero
0b0010: very large value to be divided by value close to zero
0b1000: indicates both operands are close to zero

See also: **environ.kernel/closetozero** setting.

`environ.arity (f)`

The function returns the number of parameters of a function `f` and additionally a Boolean indicating whether its parameter list includes a `` `?` `` (varargs) token or not, plus the number of upvalues used in functions created by factories. The first return does not count a varargs token.

See also: **debug.getinfo**.

`environ.attrib (obj)`

The function returns various internal status information on structures and procedures.

With the table `obj`, returns a new table with

- the current maximum number of key~value pairs allocable to the array and hash parts of `obj`; in the resulting table, these values are indexed with keys `'array_allocated'` and `'hash_allocated'`, respectively,
- the number of key~value pairs actually assigned to the respective array and hash sections of `obj`; in the resulting table, these values are indexed with keys `'array_assigned'` and `'hash_assigned'`,
- an indicator `'array_hasholes'` stating whether the array part contains at least one hole,
- an indicator `'bytes'` stating the estimated number of bytes reserved for the structure,
- an indicator `'metatable'` denoting whether a metatable has been attached to the structure,
- if present, a user-defined type is indexed by the `'utype'` key, otherwise **fail**,
- if present, a weak table is indexed by the `'weak'` key, otherwise **fail**,
- the `'length'` entry contains the estimated number of elements in a table (see **tables.getsize** & **tables.getsizes**),
- the `'lowest'` and `'keys'` represent the lowest and highest index positions in the array part; they are set to zero if there is no array part in the table;
- the `'dummynode'` entry indicates whether a table has no allocated hash part.
- See also: **tables.borders**, **tables.indices**, **tables.maxn**.

With the set `obj`, returns a new table with

- the current maximum number of items allocable to the set; in the resulting table, this value is indexed with the key `'hash_allocated'`.
- the number of items actually assigned to `obj`; in the resulting table, this value is indexed with the key `'hash_assigned'`,
- an indicator `'bytes'` stating the estimated number of bytes reserved for the structure,
- an indicator `'metatable'` betoking whether a metatable has been attached to the structure,
- if present, a user-defined type is indexed by the `'utype'` key, otherwise **fail**.

With the sequence `obj`, returns a new table with

- the maximum number of items assignable; in the resulting table, this value is indexed with the key `'maxsize'`. If the number of entries is not restricted, `'maxsize'` is **infinity**.
- the current number of items actually assigned to `obj`; in the resulting table, this value is indexed with the key `'size'`,
- an indicator `'bytes'` stating the estimated number of bytes reserved for the structure,

- an indicator `'metatable'` betoking whether a metatable has been attached to the structure,
- if present, a user-defined type is indexed by the `'utype'` key, otherwise **fail**,
- if present, a weak table is indexed by the `'weak'` key, otherwise **fail**.

With the register `obj`, returns a new table with

- the total number of items assigned; in the resulting table, this value is indexed with the key `'size'`.
- the current top indexed by the key `'top'`,
- an indicator `'bytes'` stating the estimated number of bytes reserved for the structure,
- an indicator `'metatable'` indicating whether a metatable has been attached to the structure,
- if present, a user-defined type is indexed by the `'utype'` key, otherwise **fail**,
- if present, a weak table is indexed by the `'weak'` key, otherwise **fail**.

With the pair `obj`, returns a new table with

- an indicator `'bytes'` stating the estimated number of bytes reserved,
- an indicator `'metatable'` betoking whether a metatable has been attached to the structure,
- if present, a user-defined type is indexed by the `'utype'` key, otherwise **fail**,
- if present, a weak table is indexed by the `'weak'` key, otherwise **fail**.

With function `obj` returns a new table with

- the information whether the function is a C or an Agena function. In the resulting table, this value is indexed with the key `'C'`;
- the information whether a function contains a remember table, indicated by the key 'rtableWritemode', where the entry **true** indicates that it is an rtable (which is updated by the **return** statement), where **false** indicates that it is an rotable (which cannot be updated by the **return** statement), and where **fail** indicates that the function has no remember table at all,
- the information whether an internal storage table is present, in the `'storage'` field, see Chapter 6.2.5,
- an indicator `'bytes'` stating the estimated number of bytes reserved,
- if present, a user-defined type is indexed by the `'utype'` key, otherwise **fail**,
- the number of parameters excluding varargs (?) in the `'arity'` field (with OOP methods, the result includes the **self** variable),
- a Boolean indicating whether the varargs tokern (?) is part of the parameter list, in the `'varargs'` field,
- the number of upvalues in the `'nupvals'` field.

`environ.decpoint ()`

Returns the decimal point separator used in the current locale. It is an alternative to the expression **os.getlocale.decimal_point**, but is faster.

`environ.gc ([opt [, arg]])`

This function is a generic interface to the garbage collector. It performs different functions according to its first argument, `opt`:

- **'stop':** stops the garbage collector.
- **'restart':** restarts the garbage collector.
- **'collect':** performs a full garbage-collection cycle (if no option is given, this is the default action).
- **'count':** returns the total memory in use by Agena (in Kbytes).
- **'step':** performs a garbage-collection step. The step 'size' is controlled by `arg` (larger values mean more steps) in a non-specified way. If you want to control the step size you must experimentally tune the value of `arg`. Returns **true** if the step finished a collection cycle.
- **'setpause':** sets `arg`/100 as the new value for the pause of the collector.
- **'setstepmul':** sets `arg`/100 as the new value for the step multiplier of the collector.
- **'status':** determines whether the garbage collector is running or has been stopped, and returns **true** - i.e. collection has been activated - or **false**.

`environ.getfenv (f)`

Returns the current environment in use by the function. `f` can be an Agena function or a number that specifies the function at that stack level: Level 1 is the function calling **getfenv**. If the given function is not an Agena function, or if `f` is 0, **getfenv** returns the global environment. The default for `f` is 1.

`environ.getopt (args, format)`

The function parses command-line options passed from the underlying operating system to an Agena script.

Each option (switch) may consist of exactly one letter, preceded by a dash or slash, multi-letter switches are not supported and will be incorrectly processed.

Examples:

Valid: agena script.agn -h
Valid: agena script.agn /h
Valid: agena script.agn -apx (expanded to -a -p -x)
Valid: agena script.agn /apx (expanded to /a /p /x)
Valid: agena script.agn -val 3.141592654
Valid: agena script.agn -val=3.141592654

Invalid: agena script.agn -help (would be split into the switches -h, -e, -l and -p.

The function takes the **args** system table and a `format` string denoting the switches to detect and - if found - returns the switch name without a preceding dash or slash, an optional value if given, and the index of the next **args** entry to be processed in a subsequent call. If **args** is **null**, the function simply returns.

Depending on their position in the call from the operating system, unknown options might be ignored.

Example:

```
> for switch, optarg, nextidx in environ.getopt(args, 'ab:c::d') do
>    print(switch, optarg, nextidx)
> end
```

In this example, the format string `'ab:c:p'` has the following meaning, and you can use combinations in any order:

- `'a'`   - check for just the /a or -a switch.
- `'b:'`   - check for the /b or -b switch succeeded by a mandatory value; the switch and the value may be separated by a blank or an equals sign ('=');
- `'c::'` - check for the /c or -c switch optionally succeeded by a value, both separated by a blank or an equals sign ('=');
- `'d'`   - check for just the /d or -d switch.

For an example script, check file `getopt.agn` in the `share/scripting` folder of your Agena installation.

The function is a port to a modified version of the C library function getopt.

environ.globals (f)

Determines[28] whether function `f` includes global variables (names which have not been defined local). The return is a sequence of pairs: their left-hand side the variable name of type string, the right-hand side the respective line number (of type number). If no global variables could be found, the function returns **null**.

environ.isequal (obj1, obj2)

Compares two objects `obj1`, `obj2` for equality and returns **true** or **false**. Note that the function considers two structures (tables, sequences, registers and pairs) a and b of the same type to be different if they do not reference one another. Thus, for example, with a := [1] and b := [1], the function returns **false**, whereas a and b with a := [1] and b := a are equal.

See also: =, == operators.

---

[28] Note that the function not always returns all global names.

`environ.isselfref (obj)`

Checks whether a structure `obj` (table, set, sequence, or pair) references to itself. It returns **true** if it is self-referencing, and **false** otherwise.

The function is written in Agena and included in the lib/library.agn file.

`environ.kernel ([setting])`

`environ.kernel (setting:value)`

Queries or defines kernel settings that cannot be changed or deleted automatically by the **restart** statement.

In the first form, by passing the given `setting` as a string, the current configuration will be returned. If no argument is given, then all current settings are returned in a table.

In the second form, by passing a pair of the form `setting:value`, where `setting` is a string and `value` the respective setting given in the table below, the kernel is set to the given configuration.

The return is the new configuration.

Settings are:

Setting	Value	Description
`'alignable'`	**true** or **false**	Indicates whether your system aligns data along the 4- or 8-byte word boundary. The check is done at runtime, the setting is not compiled into the interpreter.
`'bitsint'`	a number	Number of bits in a C int, should mostly be 32.
`'buffersize'`	a number	The default buffer size for file operations for the **os.fcopy**, **net.receive**, and **binio.readlines** functions. Must be set to [512 .. 1024^3] It is equal to the C constant BUFSIZ in stdio.h. Grep LUAL_BUFFERSIZE in the C sources.
`'builtoncpu'`	a string	Denotes the CPU architecture Agena has been built on.
`'clockspersec'`	an integer	Ticks per second measured by the **os.clock** function, read-only.
`'closetozero'`	a number	threshold so that the binary **inc**, **dec**, **mul**, **div**, **intdiv** and **mod** operators can recognise operators close to zero. Default is **DoubleEps**.

Setting	Value	Description
`'constants'`	**true** or **false**	Turn constants feature on or off. Default: on.
`'debug'`	**true** or **false**	Prints further debugging information if the initialisation of a C dynamic library failed
`'digits'`	an integer in [1, 17]	Sets the number of digits used in the output of numbers. Note that this setting does not affect the precision of arithmetic operations. The default is 14.
`'duplicates'`	**true** or **false**	Turns duplicate declarations (shadowing) warnings on (**true**) and off. Default: **true**.
`'emptyline'`	**true** or **false**	If set **true** (the default), two input regions are always separated by an empty line. If set **false**, no empty line is inserted.
`'errmlinebreak'`	a positive integer	Stores the maximum number of characters to be displayed per line in syntax error messages. Default is 70.
`'eps'`	a number	Stores the accuracy threshold epsilon used by the ~= operator and the **approx** function.
`'doubleeps'`	a number	Gets and sets **DoubleEps** variable; default is 2.2204460492503131e-16.
`'foradjust'`	**true** or **false**	Controls auto-correction of iteration values x close to zero in numeric **for** loops with fractional step sizes. Default is **true**, i.e. x will be set to zero if x < hEps.
`'glibc'`	a number	When Agena has been compiled with GCC, denotes the major GLIBC version linked during compilation. Otherwise is **null**.
`'glibcminor'`	a number	Dito, but with respect to the minor GLIBC version.
`'gui'`	**true** or **false**	If set **true**, tells the interpreter that it has been invoked by AgenaEdit. Default is **false**.
`'hEps'`	a number	Gets and sets **hEps** variable; default is 1.4901161193847656e-12. Also controls auto-correction features in numerical for loops, see Chapter 5.2.2.
`'iso8601'`	**true** or **false**	If set to **true**, **os.date** determines weekdays according to the ISO 8601 norm. Default is **true**.
`'is32bit'`	**true** or **false**	If set to **true**, then Agena has been compiled in 32-bit mode.
`'is64bit'`	**true** or **false**	If set to **true**, then Agena has been compiled in 64-bit mode.
`'isARM'`	**true** or **false**	If set to **true**, then Agena is running on an ARM CPU.
`'isIntel'`	**true** or **false**	If set to **true**, then Agena is running on an Intel-compatible CPU.

Setting	Value	Description
`'isDOS'`	**true** or **false**	If set to **true**, then Agena is running in DOS.
`'isLinux'`	**true** or **false**	If set to **true**, then Agena is running in Linux.
`'isMac'`	**true** or **false**	If set to **true**, then Agena is running in Mac OS X.
`'isOS/2'`	**true** or **false**	If set to **true**, then Agena is running in OS/2 and successors.
`'isSolaris'`	**true** or **false**	If set to **true**, then Agena is running in Solaris.
`'isWindows'`	**true** or **false**	If set to **true**, then Agena is running in Windows.
`'kahanbabuska'`	**true** or **false**	If set to **true**, Kahan-Babuška round-off error prevention in numeric **for** loops instead of the original Kahan algorithm. Default is **false**.
`'kahanozawa'`	**true** or **false**	If set to **true**, Kahan-Ozawa round-off error prevention in numeric **for** loops instead of the original Kahan algorithm. Default is **false**.
`'lastcontint'`	a number	Largest accurately representable integer (usually 2^{53}).
`'libnamereset'`	**true** or **false**	If set **true**, the **restart** statement resets **libname** and **mainlibname** to their original values. Default is **false**.
`'skipagenapath'`	**true** or **false**	If set to **true**, the interpreter has been started with the -a option, ignoring the setting of the AGENAPATH environment variable. Default is **false**.
`'loaded'`	a set	Returns the names of all basic libraries initialised at start-up of the interpreter.
`'longtable'`	**true** or **false**	If set **true**, then each key~value pair in a table will be printed at a separate line, otherwise a table will be printed like sets or sequences. Default is **false**.
`'maxinteger'`	a number	Maximum representable 4-byte signed integer used internally
`'maxlong'`	a number	Maximum value of a signed 32-bit integer, usually 2,147,483,647.
`'maxulong'`	a number	Maximum value of an unsigned 32-bit integer, usually 4,294,967,295.
`'minlong'`	a number	Minimum value of a signed 32-bit integer, usually -2,147,483,648.
`'nbits'`	a number	Number of bits in a 4-word integer.
`'nbits64'`	a number	Number of bits in an 8-word integer.
`'pathmax'`	a string	Returns the maximum path length accepted by the operating system, an integer.
`'pathsep'`	a string	The token that separates paths in libname;

Setting	Value	Description
		by default is ';', cannot be changed. Grep LUA_PATHSEP in the C sources.
`'promptnewline'`	**true** or **false**	If set to **true**, prints an empty line between the input and output regions. Default is **false**.
`'readlibbed'`	a set	Returns the names of all libraries manually imported in a session by **import**, **readlib**, **initialise**.
`'regsize'`	a number	Sets the default size of registers, the number must be a non-negative integer.
`'rounding'`	a string	Returns or sets the current rounding method (beware, this may cause unwanted results; see also **math.rint**): `'downward'` rounds down to the next lower integer, `'upward'` rounds up to the next greater integer, `'nearest'` rounds up or down toward whichever integer is nearest (default on most systems), `'zero'` rounds toward zero.
`'seqautoshrink'`	**true** or **false**	If set to **false**, Agena does not free memory when you remove sequence values; default is **true**.
`'signedbits'`	**true** or **false**	If set to **true**, the bitwise operators **&&**, ~~, ||, ^ ^, <<<, >>>, <<<< and >>>> internally use signed integers (the default), otherwise they use unsigned integers.
`'skipinis'`	**true** or **false**	If set to true, does not read the Agena initialisation files agena.ini / .aganea.init at **restart**. Default is **false**.
`'skipmainlib'`	**true** or **false**	If set to true, does not read the main library file **lib/library at restart**. Default is **false**.
`'smallestnormal'`	a number	Smallest normal number (usually 2^{-1022}).
`'warnings'`	an integer or Boolean	When just reading this setting, returns a number: 0 - warning system is off; 1 - ready to start a new message; 2 - previous message is to be continued. Setting the warning mode requires a Boolean: **true** - switch on warnings, **false** - switch them off.
`'zeroedcomplex'`	**true** or **false**	When set to **true**, real and imaginary parts of complex values close to zero are rounded to zero on output. (Note that internally, complex values are not rounded.) Default is **false**.

Examples:

```
> environ.kernel('signedbits'):
false

> environ.kernel(signedbits = true):
true
```

See also: **environ.system**.

`environ.onexit ()`

If assigned a function to the name **environ.onexit**, this function is automatically called when quitting or restarting Agena. For more information, see **bye**.

`environ.pointer (obj)`

Converts `obj` to a generic C pointer (void*) and returns the result as a string. `obj` may be userdata, a table, set, sequence, register, pair, thread or function; otherwise, **pointer** returns **fail**. Different objects will give different pointers, which gives a unique string identifier.

`environ.ref (tbl, obj [, option])`

Creates a unique integer reference for any argument `obj` and inserts `obj` into table `tbl` at position ref (i.e. `tbl`[ref] :=`obj`).

The function returns ref. Do not manually put any integer keys into `tbl` or delete them, always use **environ.ref** and **environ.unref** to modify `tbl`.

By default, `obj` is always inserted into `tbl`, even if it is already stored there.

If the optional third argument is `'reference'` or `'full'`, then a check is performed to ensure that `obj` has not already been included in `tbl`. If `obj` is already in `tbl`, it is not inserted again and the integer index of `obj` in `tbl` is simply returned.

If `option` is `'reference'`, the function uses **environ.isequal** for the check, whereas with the option `'full'`, the standard = equality operator is being used.

See also: **environ.unref**, **sema.open**, **utils.uuid**.

`environ.setfenv (f, table)`

Sets the environment to be used by the given function. `f` can be an Agena function or a number that specifies the function at that stack level: Level 1 is the function calling **setfenv**. **setfenv** returns the given function.

As a special case, when `f` is 0 **setfenv** changes the environment of the running thread. In this case, **setfenv** returns no values.

`environ.system ()`

Returns a table with the following system information: The size of various C types (char, int, long, long long, float, double, long double, uint16_t, int32_t, int64_t), the smallest and largest numeric values for C doubles (fields 'mindouble', 'maxdouble'), C unsigned short int (16-bit, 'maxushort'), C long ints (32-bit, fields 'minlong', 'maxlong'), C long long ints (fields 'minlonglong', 'maxlonglong'), and C unsigned long ints (field 'maxulong') (all compiled into the Agena binary), the number of bits in an C unsigned char ('bitschar') and a 32-bit integer (field 'bitsint'), the endianness of your platform (field 'endianess'), the hardware (field 'hardware') and the operating system (field 'OS') for which the Agena executable has been compiled. The 'floatradix' key represents the GNU C FLT_RADIX environment variable, with usually is 2.

See also: **environ.kernel**.

`environ.unref (tbl, ref)`

With `tbl` a table and `ref` an integer, deletes value `tbl`[`ref`] and returns it. See also: **environ.ref**.

`environ.used ( [opt] )`

By default, returns the total memory in use by Agena in Kbytes. If `opt` is the string `'bytes'`, `'kbytes'`, `'mbytes'`, `'gbytes'` or `'tbytes'`, the number will be returned in the given unit.

See also: **os.freemem**, **os.memstate**.

`environ.userinfo (f, level [, ···])`

Writes information to the user of a procedure `f` depending on the given `level`, an integer. The information to be printed is passed as the third, etc. arguments and may be either numbers or strings.

At first the procedure should be registered in the **environ.infolevel** table along with a `level` (an integer) indicating the infolevel setting at which information will be printed, e.g. `environ.infolevel[myfunc] := 1`.

If you do not enter an entry for the function to the **environ.infolevel** table, then nothing is printed.

```
> f := proc(x) is
>    environ.userinfo(f, 1, 'primary info to the user:     ', x, '\n');
>    environ.userinfo(f, 2, 'additional info to the user: ', x, '\n')
> end;
```

If the `level` argument to **userinfo** is equal or less than the **environ.infolevel** table setting, then the information is printed, otherwise nothing is printed.

```
> environ.infolevel[f] := 2;

> f('hello !');
primary info to the user:     hello !
additional info to the user:  hello !
```

Now the infolevel is decreased such that less information will be output.

```
> environ.infolevel[f] := 1;

> f('hello !');
primary info to the user:     hello !
```

See also: **environ.warn**.

environ.warn (str [, ···])

Emits a warning with a message composed by the concatenation of all its arguments (which should be strings or numbers).

By convention, a one-piece message starting with '@' is intended to be a control message, which is a message to the warning system itself. In particular, the standard warning function in Lua recognizes the control messages "@off", to stop the emission of warnings, and "@on", to (re)start the emission; it ignores unknown control messages.

If called without arguments, returns a Boolean indicating whether the warning system is on or off, and the current warning state as an integer:

- 0 - warning system is off;
- 1 - ready to start a new message;
- 2 - previous message is to be continued.

See also: **environ.infolevel**, **environ.userinfo**.

14.3 package - Modules

The package library provides a basic facility to inspect which packages have been loaded in a session.

`package.checkclib (pkg)`

Checks whether the package denoted by the string `pkg` and stored to a C dynamic library has already been initialised. If not, it returns a warning printed on screen and creates an empty package table. Otherwise it does nothing.

`package.loadclib (packagename, path)`

Loads the C library `packagename` (with extension `.so` in UNIX and Mac, or `.dll` in Windows) residing in the folder denoted by `path`. `path` must be the name of the folder where the C library is stored, and not the absolute path name of the file. The function returns **true** in case of success and **false** otherwise. On successful initialisation, the name of the package is entered into the **package.readlibbed** set.

See also: **readlib**, **with**.

`package.loaded`

A table containing all the names of the packages that have been initialised, either at start-up or later on in a session.

`package.packages ()`

Returns a set with all the names of Agena's standard libraries that are initialised at start-up. The set does not include packages that have been loaded manually later on in a session, for example by the **import** statement or the **readlib** or with functions.

`package.readlibbed`

A set with all the names of the packages that have been initialised with the **readlib** and **with** functions, and the **import** statement. This set may be deprecated in future versions of Agena.

14.4 rtable - Remember Tables

This package comprises functions to administer remember tables.

rtable.defaults (f)

rtable.defaults (f, tab)

rtable.defaults (f, null)

Administrates read-only remember tables of functions. As it works exactly like the **remember** function, except that it creates remember tables that cannot be updated by the **return** statement, please refer to the description of the **rtable.remember** function for further details.

rtable.remember (f)

rtable.remember (f, tab)

rtable.remember (f, null)

Administers remember tables.

In the first form, the remember table stored to procedure `f` will be returned. See **rtable.get** for more information.

In the second form, **remember** adds the arguments and returns contained in table `tab` to the remember table of function `f`. If the remember table of `f` has not been initialised before, **remember** creates it. If there are already values in the remember table, they are kept and not deleted.

If `f` has only one argument and one return, the function arguments and returns are passed as key~value pairs in table `tab`.

If `f` has more than one argument, the arguments are passed in a table. If `f` has more than one return, the returns are passed in a table, as well.

Valid calls are:

```
import rtable alias remember;

remember(f, [0 ~ 1]);           # one argument 0 & one return 1
remember(f, [[1, 2] ~ [3, 4]);  # two arguments 1, 2 & two returns 3, 4
remember(f, [1 ~ [3, 4]]);      # one argument 1 & two returns 3, 4
remember(f, [[1, 2] ~ 3]];      # two arguments 1, 2 & one return 3
```

In the third form, by explicitly passing **null** as the second argument, the remember table of `f` is destroyed and a garbage collection run to free up space occupied by the former rtable.

remember always returns **null**. It is written in Agena and included in the lib/library.agn file.

See Chapter 6.18 for examples. See also: **rtable.defaults**.

rtable.forget (f)

Empties the remember table or read-only remember table of procedure `f` entirely, giving back the space formerly occupied to the interpreter. It also enforces an immediate garbage collection. The function returns **null**.

rtable.get (f [, option])

Returns the contents of the current remember table or read-only remember table of procedure `f`. If any value for `option` is given, the internal remember table including all the hash values are returned.

```
> fib := proc(n) is
>    assume(n >= 0);
>    return fib(n - 2) + fib(n - 1)
> end;

> rtable.remember(fib, [0 ~ 0, 1 ~ 1]);

> rget(fib):
[[0] ~ [0], [1] ~ [1]]
```

You cannot destroy the internal remember table by changing the table returned by **rtable.get**.

rtable.init (f)

Creates a remember table (an empty table) for procedure `f`. The procedure must have been written in Agena; reminisce that rtables for C API functions are not supported and that in these cases the function quits with an error.

If there is already a remember function for `f`, it is overwritten. **init** returns **null**.

rtable.mode (f)

Returns the string `'rtable'` if function `f` has a remember table, `'rotable'` if f has a read-only remember table (that cannot be updated by the **return** statement), and the string `'none'` otherwise.

rtable.purge (f)

Deletes the remember table or read-only remember table of procedure `f` entirely. It also enforces an immediate garbage collection. The function returns **null**.

rtable.roinit (f)

Creates a read-only remember table (an empty table) for procedure `f`, which may be either a C function or an Agena procedure.

If there is already a remember function for `f`, it is overwritten. **roinit** returns **null**.

rtable.put (f, arguments, returns)

The function adds one (and only one) function-argument-and-returns `pair` to the already existing remember table or read-only remember table of procedure `f`.

`arguments` must be a table array, `returns` must also be a table array. If the argument(s) already exist(s) in the remember table, then the corresponding result(s) are replaced with `returns`.

Given a function `f := << x -> x >>` for example, valid calls are:

```
rtable.put(f, [1], [2]); rtable.put(f, [1, 2], [2]);
rtable.put(f, [1], [1, 2]).
```

14.5 registry - Access to the Registry

This package provides limited access to the registry (see Chapter 6.31). It should be used carefully.

Its library functions are:

`registry.anchor (key, value)`

Inserts a new key ~ value pair into the registry, where the `key` is a unique string, and the `value` the corresponding data. To delete entries in the registry, pass **null** for `value`.

The function returns nothing.

See also: **registry.get**.

`registry.anyid (key)`

The function has become obsolete and just returns the string `key` given.

See also: **registry.get**, **utils.uuid**.

`registry.get (key)`

The function returns the registry value indexed by `key`, which may be any type. If the registry entry is occupied by userdata, refers to loaded libraries or open files, the function just returns **null**. Otherwise, the entry is simply returned.

If a C library metatable contains the __metatable read-only `metamethod`, **null** will be returned, as well.

With metatables defined by C libraries, it is still possible to delete or change metamethods, so extreme care should be taken when referencing to metatables returned. Especially, the __gc metamethod must not be deleted or changed.

See also: **registry.anchor**.

14.6 stack - Built-In Number and Character Stacks

The functions and operators in this package work with one of the six built-in numerical and character stacks, internal last-in-first-out data structures that can either store numbers or characters only. The stacks are addressed by their stack number and not variable names.

Stacks 1 to 3 are number stacks, stacks 4 to 6 are character stacks. The default stack is stack 1, i.e. a number stack.

Never use or change numeric stack 0 which is used by Agena internally as some sort of fixed-sized register array for numeric and other operations.

You may switch between the stacks by calling switchd(1) for the first stack, switchd(2) which is the second stack, etc.

While any element in - for example - a sequence occupies 24 bytes of memory, a number in a numeric stack takes only eight bytes, and a character in a character stack only one byte.

You can push a theoretically unlimited amount of numbers or characters onto the currently selected stack, with 128 pre-allocated slots after Agena initialisation.

If more numbers or characters are added, Agena automatically enlarges the current stack. However, if values are removed, Agena itself does not shrink the allocated memory. Call **stack.shrinkd** to accomplish this if required, and **stack.sized** to query the amount of used and internally allocated space.

Example: to convert a decimal number into the binary system, you might use a numerical stack:

```
> tobinary := proc(x :: number) is  # for positive numbers only
>    local base := 2;      # new base
>    local r := '';
>    stack.resetd();       # always clear stack before usage, destructive !
>    while x > 0 do
>       pushd(x % base);   # push the remnant onto the stack (0 or 1)
>       x := x \ base
>    od;
>    while allotted() do   # now traverse the stack from top to bottom
>       r := r & popd()    # and also remove the remnants one after another
>    od;
>    return r             # return result as a string
> end;

> switchd(1): # select a numeric stack

> tobinary(6):
110
```

The names of functions and operators usually end in a final `d`.

The basic library functions and operators are:

`allotted ([n])`

If given no argument and if the stack contains one or more elements, the function returns the number of values in the stack, else it returns **null**. It can be easily used in **while** loops to traverse a stack, see example above.

If given a stack number `n`, it either returns the number of elements currently in the given stack or **null** if no values are in this stack.

See also: **stack.sized**.

`cell (idx)`

The operator returns the element stored at the absolute or relative position `idx` of the current stack. While -1 refers to the top of the stack, -2, to the element just below the top, etc., 0 refers to the first element at the bottom of the stack, 1 one step `above`, etc. If the stack is empty or `idx` is out-of-bounds, **null** will be returned.

`popd ([n [, anyoption]])`

The function pops `n` numbers or characters from the top of the current stack, or if `n` is not given, the element at the current top of the stack. If the stack is empty, **null** will be returned.

By default, the function returns the last value popped. This can be suppressed if any second argument is given.

See also: **pushd**.

`pushd (x [, ...])`
`pushd (s)`

In the first form, the statement pushes one or more numbers, strings or single characters `x` onto the current stack. In the second form, the command pushes all the numbers, strings or single characters in the sequence `s`. With strings, all the characters in the string are pushed, with the last character put at the stack top, and all the preceding characters below.

The function returns nothing.

You may or may not enclose the arguments in brackets.

If the current stack is a character stack and `x` or an element in `s` is an integer in the range 0 to 9, both inclusive, the operator converts `x` to a character and inserts this character into the stack. If you insert an empty string and dump the string using - for

example - **stack.dumpd** later, the resulting string is terminated at the position you inserted the empty string.

Hint: If the very last argument is the pair `'stack':<stacknumber>`, with <stacknumber> an integer, the values are pushed onto the given stack <stacknumber>. Using this option, however, may slow down the operation since processing an option takes quite some computation time.

See also: **stack.pushstringd**.

switchd (n)

The statement switches to stack `n` with `n` an integer.

You may not or may not enclose the argument in brackets.

See also: **stack.selected**, **stack.switchto**.

The **stack** library features the following auxiliary procedures:

stack.attribd ()

Returns various status information on the current stack in a table with the keys: `'currentstack'` (number of the current stack), `'defaultsize'` (number of default slots), `'isnumstack'` (**true** if the stack stores Agena numbers, **false** if the stack stores ASCII character codes), `'stackmax'` (maximum number of allocated slots), `'stacktop'` (current stack top, counting from 0).

stack.choosed ()

Returns the ID of both the number and character stack with the least number of values in them. The first return is the ID of the smallest number stack, the second return the one of the smallest character stack.

See also: **switchd**, **stack.selected**.

stack.dequeued ()

Removes the value at the bottom of the current stack, returns it, and moves all elements to close the space. It is equivalent to **stack.removed**(0).

See also: **stack.enqueued**.

`stack.dumpd ([n] [, option])`

With a numeric stack, returns all values in the current stack in a new sequence if given no argument, or the last n values pushed onto the stack, and pops them all from the stack. The number of pre-allocated slots is not changed, see **stats.shrinkd**.

With a character stack, when given no argument, all characters are returned as exactly one string, first-in first-out style. If n is given, the last n characters inserted are returned in exactly one string.
If passed **true** as the very last argument, the selected numbers or characters are returned in reverse order. This saves an expensive call to **stack.reversed**.

If the stack is empty, the function just returns **null**.

See also: **stack.explored**.

`stack.enqueued (x)`

Inserts the number or character x at the bottom of the current stack, shifting up other elements to open space. The function returns nothing. It works like **stack.insertd**(0, x).

See also: **stack.dequeued**.

`stack.explored ()`

Returns the entire contents of the current stack without modifying it. If the current stack is numeric, the return is a sequence of numbers, otherwise a string will be returned.

If the stack is empty, the function just returns **null**.

See also: **stack.dumpd**.

`stack.insertd (idx, x)`

Inserts the number or character x at the given relative stack position idx, shifting up other elements to open space. The function returns nothing.

If idx is a non-negative integer, 0 represents the bottom of the stack, 1 the position just above the bottom, etc. If idx is negative, -1 represents the stack top, -2 the element just below the top, etc. If no argument is given, idx is set to -1 by default, i.e. the top element.

See also: **stack.dequeued**, **stack.enqueued**, **stack.removed**.

```
stack.mapd ([idx, ] f, [···] [, true])
stack.mapd (true, f, [, true])
```

In the first form, if `idx` is not given or if `idx` = -1, the default, maps function `f` on the top element of the current character or numeric stack. With a character stack, the result of the call to `f` must be a character, and with a number stack the result must be a number, otherwise the function issues an error.

If `idx` is a negative integer, the **stack.mapd** applies `f` on the |`idx`|-th value in the stack.

If `f` is a multivariate function, its second, third, etc. argument must be passed right after `f`.

In the second form, by passing the Boolean **true** and a univariate or multivariate function `f` of n parameters, **stack.mapd** takes the n top stack values, passes them as arguments to `f`, with the value at the top of the stack the last argument, and returns the result. If the parameter list of `f` contains a ?-varargs token, then all stack elements are passed to `f`. Example:

We have a function of three parameters x, y, z that returns x*y + z. The call puts the top three argumements 3, 2, 4

```
> pushd 3, 2, 4
```

into the argument list, pops them from the stack, pushes the result of the function call onto the stack and returns it:

```
> stack.mapd(true, << x, y, z -> fma(x, y, z) >>):
10
```

In both the first and the second form, the stack element(s) is (are) replaced by the result of the function call if **true** is not given as the last argument, but the rest of the stack is left unchanged. Otherwise, if you pass **true**, the stack element(s) is (are) not replaced and the result is just put onto the stack top.

```
stack.pushstringd (str)
```

Pushes all the characters in string `str` onto a character stack. After the operation, `str`[-1] resides at the top of the stack, `str`[-2] just below, etc. The return is the size of `str`, i.e. **size** `str`.

See also: **pushd**.

```
stack.pushvalued (idx)
```

Pushes the value residing at the given relative stack position `idx` to the top of the stack. The original value also remains at position `idx`.

If `idx` is a non-negative integer, 0 represents the bottom of the stack, 1 the position just above the bottom, etc. If `idx` is negative, -1 represents the stack top, -2 the element just below the top, etc. If no argument is given, `idx` is set to -1 by default, i.e. the top element.

`stack.removed (idx)`

Removes the value residing at the given relative stack position `idx`, returns it, and moves all elements to close the space.

If `idx` is a non-negative integer, 0 represents the bottom of the stack, 1 the position just above the bottom, etc. If `idx` is negative, -1 represents the stack top, -2 the element just below the top, etc. If no argument is given, `idx` is set to -1 by default, i.e. the top element.

See also: **stack.insertd**, **stack.dequeued**, **stack.enqueued**.

`stack.readbytes (filehandle [, bytes] [, options])`

By default, the function reads **environ.kernel('buffersize')** bytes from the file denoted by `filehandle` and writes them to the current stack.

Open the file with **binio.open** before the first call and when you are finished, call **binio.close** to release the file.

The function increments the file position thereafter so that the next bytes in the file can be read with a new call to various **stack.readbytes**.

You may change the kernel buffer size value to any other values in order to read less or more bytes.

If `bytes` is given, the function writes `bytes` bytes from the file denoted by `filehandle` to the current stack.

To write the file contents to another than the current stack, pass the optional `stack` option with the preferred stack number, e.g.:

```
> stack.readbytes(fh, stack=6);
```

By default, the function ignores any newlines (ASCII 10) or carriage returns (ASCII 13) in the file. You can change this by setting the `ignore` option and passing a string of explicit characters that shall be skipped, e.g.:

```
> stack.readbytes(fh, ignore=" .\n");  # skip white space, dot & newline
```

By default the function reads in embedded zeros and treats them as every other byte. If you pass the `eof` option and set it to **true**, then the function quits if it

encounters an embedded zero in the file. The file pointer is automatically reset to the position of the embedded zero. The default is **false**, i.e. the whole file is read in.

The function returns both the number of bytes read from the file and the number of bytes written to the stack. If the end of the file has been reached, **null** will be returned. In case of an error, it quits with the respective error.

See also: **binio.readbytes**.

`stack.replaced (idx, x)`

Replaces the value at stack position `idx` with the number or one-character string `x`. The value replaced will be returned.

If `idx` is a non-negative integer, 0 represents the bottom of the stack, 1 the position just above the bottom, etc. If `idx` is negative, -1 represents the stack top, -2 the element just below the top, etc. If no argument is given, `idx` is set to -1 by default, i.e. the top element.

Note that with character stacks, if `x` should be the empty string, the string returned by **strings.dumpd** will be terminated at position `idx`.

`stack.resetd ([···])`

If given no arguments, clears the entire stack so that it becomes empty. The functions does not return anything and should be used cautiously as another function might still need elements in the stack.

If passed one or more valid stack numbers (integers), the function conducts the same for the given stack(s).

The number of pre-allocated slots is not reset, however, see **stack.shrinkd**.

`stack.reversed ([n])`

If `n` is not given, reverses the positions of all the elements in the current stack. If `n` is given, only the last `n` elements pushed onto the stack are reversed. The function returns nothing.

See also: **strings.reverse**.

`stack.rotated ([n])`

Moves all the elements in the current stack `n` places from the bottom to the top if `n` is positive, and `n` places from the top to the bottom if `n` is negative. The default is `n` = +1. The function returns nothing.

`stack.selected ()`

Returns the number of the currently selected stack, i.e. an integer.

See also: **switchd**, **stack.choosed**.

`stack.shrinkd ([any])`

With no argument, shrinks the number of pre-allocated but not assigned slots in the current stack if possible. If any argument is given, all stacks are shrunk.

The function returns the new number(s) of pre-allocated slots but does not pop any elements. The function is useful to reduce memory consumption if a lot of values have been removed from the stack.

See also: **stack.resetd**.

`stack.sized ()`

Returns the current number of elements in the current stack along with the current maximum number of slots internally allocated by the system. See also: **allotted**.

`stack.sorted ([n])`

Sorts all or the last `n` values pushed onto the current stack in ascending order using the fast Introsort algorithm. If no argument is passed, all elements in the current stack are processed. The function returns nothing.

The function works with numeric stacks only.

`stack.swapd (i, j)`

Swaps the values stored at position `i` and `j` of the current stack.

`stack.switchto ([x])`

Determines the current stack number p, and then switches to the number stack q with the least number of elements if any number `x` or no argument is given, or to the character stack q with the least number of elements if any string `x` is given.

The function returns p and q, in this order.

You can use p later to switch back to the former stack with the **switchd** statement.

The following functions perform real arithmetic on a numeric stack:

`stack.addtod (x [, true])`

Adds its argument x, a number, to the top element of the current numeric stack.

The top stack element is replaced by the resulting sum if **true** is not given as the last argument, but the rest of the stack is left unchanged. Otherwise, if you pass **true**, the top stack element is not replaced and the result is just put onto the stack top.

The function also returns the sum computed. It uses the Kahan-Babuška algorithm to prevent round-off errors.

See also: **stack.mulby**, **stack.powd**, **stack.sumupd**.

`stack.mulbyd (x [, true])`

Multiplies its argument x, a number, by the top element of the current numeric stack.

The top stack element is replaced by the resulting product if **true** is not given as the last argument, but the rest of the stack is left unchanged. Otherwise, if you pass **true**, the top stack element is not replaced and the result is just put onto the stack top.

The function also returns computed product.

See also: **stack.addto**, **stack.mulupd**, **stack.powd**, **stack.recipd**.

`stack.absd ([true])`

Computes the absolute value of the top element of the current numeric stack.

The top stack element is replaced by the result if **true** is not given as an argument, but the rest of the stack is left unchanged. Otherwise, if you pass **true**, the top stack element is not replaced and the result is just put onto the stack top.

The function also returns the result.

`stack.negated ([true])`

Multiplies the top numeric stack element by -1.

The top stack element is replaced by the result if **true** is not given as an argument, but the rest of the stack is left unchanged. Otherwise, if you pass **true**, the top stack element is not replaced and the result is just put onto the stack top.

The function also returns the result.

`stack.recipd ([true])`

Computes the reciprocal of the top element of the current numeric stack.

The top stack element is replaced by the result if **true** is not given as an argument, but the rest of the stack is left unchanged. Otherwise, if you pass **true**, the top stack element is not replaced and the result is just put onto the stack top.

The function also returns the result.

`stack.intd ([true])`

Rounds the number on the top of a numeric stack to the nearest integer towards 0.

The top stack element is replaced by the result if **true** is not given as an argument, but the rest of the stack is left unchanged. Otherwise, if you pass **true**, the top stack element is not replaced and the result is just put onto the stack top.

The function also returns the result.

`stack.fracd ([true])`

Converts the number on the stack top to its fractional part.

The top stack element is replaced by the result if **true** is not given as an argument, but the rest of the stack is left unchanged. Otherwise, if you pass **true**, the top stack element is not replaced and the result is just put onto the stack top.

The function also returns the result.

`stack.intdivd (x [, true])`

Integer division of the top element of the current numeric stack and the given number x; works like the \ operator.

The top stack element is replaced by the result if **true** is not given as the last argument, but the rest of the stack is left unchanged. Otherwise, if you pass **true**, the top stack element is not replaced and the result is just put onto the stack top.

The function also returns the result.

See also: **stack.modd**.

`stack.modd (x [, true])`

Modulus of the top element of the current numeric stack and the given number x; works like the **%** operator.

The top stack element is replaced by the result if **true** is not given as the last argument, but the rest of the stack is left unchanged. Otherwise, if you pass **true**, the top stack element is not replaced and the result is just put onto the stack top.

The function also returns the result.

See also: **stack.intdivd**.

`stack.powd (power [, true])`

Raises the top element of the current numeric stack to the given `power`, a number.

The top stack element is replaced by the result if **true** is not given as the last argument, but the rest of the stack is left unchanged. Otherwise, if you pass **true**, the top stack element is not replaced and the result is just put onto the stack top.

The function also returns the result.

See also: **stack.antilogd**, **stack.rootd**.

`stack.pythad ([true])`

Computes the hypotenuse or inverse hypotenuse of the number below the stack top and the number on the top and returns the computed result. The function avoids over- and underflows and treats subnormal numbers accordingly.

The top stack element is replaced by the result if **true** is not given as an argument, but the rest of the stack is left unchanged. Otherwise, if you pass **true**, the top stack element is not replaced and the result is just put onto the stack top.

See also: **stack.hypotd**, **stack.invhypotd**.

`stack.squared ([true])`

Raises the top element of the current numeric stack to the power of 2.

The top stack element is replaced by the result if **true** is not given as an argument, but the rest of the stack is left unchanged. Otherwise, if you pass **true**, the top stack element is not replaced and the result is just put onto the stack top.
The function also returns the result.

See also: **stack.antilogd**, **stack.powd**, **stack.rootd**.

`stack.rootd (n [, true])`

Computes the non-principal `n`-th root of the top element of the current numeric stack. `n` must be an integer and is 2 by default.

The top stack element is replaced by the result if **true** is not given as the last argument, but the rest of the stack is left unchanged. Otherwise, if you pass **true**, the top stack element is not replaced and the result is just put onto the stack top.

See also: **stack.cbrtd**, **stack.powd**, **stack.sqrtd**.

`stack.cbrtd ([true])`

Computes the cubic root of the top element of the current numeric stack.

The top stack element is replaced by the result if **true** is not given as an argument, but the rest of the stack is left unchanged. Otherwise, if you pass **true**, the top stack element is not replaced and the result is just put onto the stack top.

See also: **stack.powd**, **stack.rootd**, **stack.sqrtd**.

`stack.sqrtd ([true])`

Computes the square root of the top element of the current numeric stack.

The top stack element is replaced by the result if **true** is not given as an argument, but the rest of the stack is left unchanged. Otherwise, if you pass **true**, the top stack element is not replaced and the result is just put onto the stack top.

See also: **stack.cbrtd**, **stack.powd**, **stack.rootd**.

`stack.fmad ([true])`

Computes $x*y + z$, where x represents the second number below the stack top, y the number below the stack top and z the number on the top and returns the computed result. The function avoids over- and underflows and treats subnormal numbers accordingly.

The top stack element is replaced by the result if **true** is not given as an argument, but the rest of the stack is left unchanged. Otherwise, if you pass **true**, the top stack element is not replaced and the result is just put onto the stack top.

`stack.hypotd ([true])`

`stack.invhypotd ([true])`

Computes the hypotenuse or inverse hypotenuse of the number below the stack top and the number on the top and returns the computed result. The function avoids over- and underflows and treats subnormal numbers accordingly.

The top stack element is replaced by the result if **true** is not given as an argument, but the rest of the stack is left unchanged. Otherwise, if you pass **true**, the top stack element is not replaced and the result is just put onto the stack top.

See also: **stack.pythad**, **stack.hypot4d**.

`stack.hypot4d ([true])`

Computes $\sqrt{x^2 - y^2}$, where x represents the number below the stack top and y the number on the top and returns the computed result. The function avoids over- and underflows and treats subnormal numbers accordingly.

The top stack element is replaced by the result if **true** is not given as an argument, but the rest of the stack is left unchanged. Otherwise, if you pass **true**, the top stack element is not replaced and the result is just put onto the stack top.

See also: **stack.hypotd**.

`stack.lnd ([true])`

Returns the natural logarithm of the top element of the current numeric stack.

The top stack element is replaced by the result if **true** is not given as an argument, but the rest of the stack is left unchanged. Otherwise, if you pass **true**, the top stack element is not replaced and the result is just put onto the stack top.

See also: **stack.antilogd**, **stack.expd**, **stack.logd**.

`stack.logd (base [, true])`

Returns the logarithm of the top element of the current numeric stack to the given `base`, with `base` a number. For the natural logarithm, pass **E** for `base`.

The top stack element is replaced by the result if **true** is not given as the last argument, but the rest of the stack is left unchanged. Otherwise, if you pass **true**, the top stack element is not replaced and the result is just put onto the stack top.

See also: **stack.antilogd**, **stack.lnd**.

`stack.antilogd (base [, true])`

Raises the given `base`, a number, to the power of the top element of the current numeric stack.

To compute the exponential function e^x, x should be the top stack element and pass **E** as function argument `base`.

The top stack element is replaced by the result if **true** is not given as the last argument, but the rest of the stack is left unchanged. Otherwise, if you pass **true**, the top stack element is not replaced and the result is just put onto the stack top.

See also: **stack.expd**, **stack.exp2d**, **stack.exp10d**.

`stack.expd ([true])`

Raises E = exp(1) to the power of the number at the stack top and returns it.

The top stack element is replaced by the result if **true** is not given as an argument, but the rest of the stack is left unchanged. Otherwise, if you pass **true**, the top stack element is not replaced and the result is just put onto the stack top.

See also: **stack.exp2d**, **stack.exp10d**.

`stack.exp2d ([true])`

Raises 2 to the power of the number at the stack top and returns it.

The top stack element is replaced by the result if **true** is not given as an argument, but the rest of the stack is left unchanged. Otherwise, if you pass **true**, the top stack element is not replaced and the result is just put onto the stack top.

See also: **stack.expd**, **stack.exp10d**.

`stack.exp10d ([true])`

Raises 10 to the power of the number at the stack top and returns it.

The top stack element is replaced by the result if **true** is not given as an argument, but the rest of the stack is left unchanged. Otherwise, if you pass **true**, the top stack element is not replaced and the result is just put onto the stack top.

See also: **stack.expd**, **stack.exp2d**.

```
stack.sind ([true])
stack.cosd ([true])
stack.tand ([true])
```

Returns the sine, cosine or tangent of the top element of the current numeric stack, in radians.

The top stack element is replaced by the result if **true** is not given as an argument, but the rest of the stack is left unchanged. Otherwise, if you pass **true**, the top stack element is not replaced and the result is just put onto the stack top.

```
stack.sinhd ([true])
stack.coshd ([true])
stack.tanhd ([true])
```

Returns the hyperbolic sine, hyperbolic cosine or hyperbolic tangent of the top element of the current numeric stack, in radians.

The top stack element is replaced by the result if **true** is not given as an argument, but the rest of the stack is left unchanged. Otherwise, if you pass **true**, the top stack element is not replaced and the result is just put onto the stack top.

```
stack.arcsind ([true])
stack.arccosd ([true])
stack.arctand ([true])
```

Returns the arcus sine, arcus cosine or arcus tangent of the top element of the current numeric stack, in radians.

The top stack element is replaced by the result if **true** is not given as an argument, but the rest of the stack is left unchanged. Otherwise, if you pass **true**, the top stack element is not replaced and the result is just put onto the stack top.

```
stack.arcsinhd ([true])
stack.arccoshd ([true])
stack.arctanhd ([true])
```

Returns the inverse hyperbolic sine, inverse hyperbolic cosine or inverse hyperbolic tangent of the top element of the current numeric stack, in radians.

The top stack element is replaced by the result if **true** is not given as an argument, but the rest of the stack is left unchanged. Otherwise, if you pass **true**, the top stack element is not replaced and the result is just put onto the stack top.

`stack.arctan2d ([true])`

Computes the arc tangent of y/x of the two top elements of the current numeric stack, in radians, where x represents the value just below the top and y the value at the top of the stack.

The top stack element is replaced by the result if **true** is not given as an argument, but the rest of the stack is left unchanged. Otherwise, if you pass **true**, the top stack element is not replaced and the result is just put onto the stack top.

`stack.meand ([n] [, true])`

Computes the arithmetic mean of all or a given number n of values in the current numeric stack and either pushes the result onto the top of the stack if **true** has been passed as the last argument; or pops all the values processed and pushes the result on top of the stack if only n or no argument has been given.

The function in general also returns the arithmetic mean.

By dividing each element before summation, the function avoids arithmetic overflows and also uses the Kahan-Babuška algorithm to prevent round-off errors during summation. Computation is done from top to bottom.

`stack.sumupd ([n] [, true])`

Sums up all or a given n number of values in the built-in numeric stack and either pushes the result onto the top of the stack if **true** has been passed as the last argument; or pops all the values summed-up and pushes the sum on top of the stack if only n or no argument has been given.

Summation is done from top to bottom. The function uses the Kahan-Babuška algorithm to prevent round-off errors during summation and also returns the sum.

`stack.mulupd ([n] [, true])`

Multiplies all or a given n number of values in the built-in numeric stack and either pushes the result onto the top of the stack if **true** has been passed as the last argument; or pops all the values multiplied and pushes the product on top of the stack if only n or no argument has been given.

Multiplication is done from top to bottom. The function in general also returns the product.

The following functions perform operations on the two values at the top of the stack, i.e. the value at the top and the value just below it. They have been deliberately kept lean for maximum performance:

stack.addtwod ()

Adds up the two numbers on top of the current stack and replaces them with the result; also returns the sum.

stack.subtwod ()

Subtracts the number on top of the current stack from the number below it and replaces them with the result; also returns the difference.

stack.multwod ()

Multiplies the number on top of the current stack by the number below it and replaces them with the result; also returns the product.

stack.divtwod ()

Divides the number below the stack top by the number on the top and replaces them with the result; also returns the quotient.

stack.intdivtwod ()

Performs an integer division of the number below the stack top by the number on the top and replaces them with the result; also returns the integer quotient.

stack.modtwod ()

Like **stack.intdivtwod** but computes the modulus.

stack.powtwod ()

Raises the number below the stack top to the power on the top and replaces both numbers with the result; also returns the power.

The following functions perform operations on a character stack:

`stack.absd ()`

Returns the ASCII value of the top element of the current numeric stack.

The function does not modify the stack.

`stack.lowerd ([true])`

Converts the character at the stack top to lower-case and returns it.

The top stack element is replaced by the result if **true** is not given as an argument, but the rest of the stack is left unchanged. Otherwise, if you pass **true**, the top stack element is not replaced and the result is just put onto the stack top.

`stack.upperd ([true])`

Converts the character at the stack top to upper-case and returns it.

The top stack element is replaced by the result if **true** is not given as an argument, but the rest of the stack is left unchanged. Otherwise, if you pass **true**, the top stack element is not replaced and the result is just put onto the stack top.

14.7 sema - Unique Identifiers

As a *plus* package, the **sema** package is not part of the standard distribution and must be activated with the **import** statement, i.e. `import sema`.

The package provides functions to create and administer unique, non-negative integer identifiers, called `semaphore ids` in this context, in a memory-saving fashion (see **sema.state** and **sema.open** for details).

There are two types of `semaphores`: a global built-in one, and one or more instances represented by userdata objects:

```
> import sema;

> sema.open():      # create a new global semaphore id
0

> s := sema.new():  # create a semaphore instance s
sema(01CCD3B8)

> sema.open(s):     # create a new semaphore id for semaphore s
0

> sema.open(s):
1

> sema.open(s):
2

> t := sema.new():  # create another semaphore instance t
sema(01CCD298)

> sema.open(t):     # create a new semaphore id for semaphore t
0
```

sema.close (id, [···])

sema.close (s, id, [···])

In the first form, closes the given global semaphore `id`, and optionally other semaphores, as well. If **sema.open** is called thereafter, it will return `id`. The function returns nothing. Note that if `id` is not the last semaphore id opened before, then Agena automatically switches to dynamic memory management consuming "O(n)" of memory space instead of using just one 8-byte integer for the internal counter.

In the second form, the `id` of semaphore `s` is closed.

See also: **sema.open**, **sema.shrink**.

sema.isopen ([s,] id)

Checks whether the given global semaphore `id` or the `id` of the semaphore `s` is open and returns **true** or **false**.

`sema.open ([s, ] [id])`

Without any argument, creates a new global semaphore id, a non-negative integer. With the very first call, returns 0, counting up by 1 subsequently as along as **sema.close** or **sema.reset** are not executed.

If the non-negative global integer id `id` is passed and `id` has not yet been created by **sema.open**, the function creates it. If `s` is being passed, then semaphore `s` returns the next free id or the given `id`.

This feature also allows you to write functions dumping the current semaphore state to a file and re-load it later, see **binio.writechar**, **binio.readchar**, **bytes.tobytes**.

If the current internal administrative memory is exceeded, the function automatically expands it. Note that if `id` is not the next semaphore id opened just before, then Agena automatically switches to dynamic memory management consuming "O(n)" of memory space instead of using just one 8-byte integer for the internal counter.

See also: **environ.ref**, **factory.count**, **utils.uuid**.

`sema.reset ([s])`

Closes all allocated semaphores ids and also shrinks the internal memory used to administer the semaphores to the default size, if possible. If semaphore `s` is given, then all ids of this semaphore are reset.

The function returns nothing. See also: **sema.shrink**.

`sema.shrink ([s])`

Without any argument, shrinks the internal memory used to administer all global semaphores ids so that it consumes the least necessary space. The same applies to semaphore `s` if given. If internal memory size could be reduced, it returns **true** and **false** otherwise. Note that due to performance reasons, **sema.close** does not try to reduce memory consumption.

See also: **sema.reset**.

`sema.state ([s])`

Returns administrative information on the current global semaphore state if not argument is given or on semaphore instance `s`, in a table.

The semaphores are internally stored using an 8 * 32 bits array by default. Each 32-bit chunk is called a `slot`.

The `'firstfreeslot'` field contains the number of the first slot that might harbour a new semaphore. Note that once you have used **sema.close**, this may just be an estimate.

The `'slot'` sequence returns the bits set in all slots, as decimal non-negative integers.

Field `'current'` is an estimate of the last allocated semaphore id.

Key `'minopen'` denotes the semaphore id with the smallest ID, `'maxopen'` the one with the largest ID.

`'nextfree'` denotes the number of the semaphore id if **sema.open** would be called next.

`'simplecounter_active'` indicates whether the memory-efficient counting is active (**true**) or dynamic memory management of the semaphore ids is in use (**false**).

`'simplecounter'` depicts the current semaphore id.

14.8 Coroutines

The operations related to coroutines comprise a sub-library of the basic library and come inside the table `coroutine`. To find out what coroutines are, please have a look at the website of the Lua programming language.

`coroutine.resume (co [, val1, ···])`

Starts or continues the execution of coroutine `co`. The first time you resume a coroutine, it starts running its body. The values `val1`, ··· are passed as the arguments to the body function. If the coroutine has yielded, resume restarts it; the values `val1`, ··· are passed as the results from the yield.

If the coroutine runs without any errors, resume returns **true** plus any values passed to yield (if the coroutine yields) or any values returned by the body function (if the coroutine terminates). If there is any error, resume returns **false** plus the error message.

`coroutine.running ()`

Returns the running coroutine, or **null** when called by the main thread.

`coroutine.setup (f)`

Creates a new coroutine, with body `f`. `f` must be an Agena function. Returns this new coroutine, an object with type 'thread'.

`coroutine.status (co)`

Returns the status of coroutine `co`, as a string: 'running', if the coroutine is running (that is, it called status); 'suspended', if the coroutine is suspended in a call to yield, or if it has not started running yet; 'normal' if the coroutine is active but not running (that is, it has resumed another coroutine); and 'dead' if the coroutine has finished its body function, or if it has stopped with an error.

`coroutine.wrap (f)`

Creates a new coroutine, with body `f`. `f` must be an Agena function. Returns a function that resumes the coroutine each time it is called. Any arguments passed to the function behave as the extra arguments to resume. Returns the same values returned by **resume**, except the first boolean. In case of error, propagates the error.

`coroutine.yield (···)`

Suspends the execution of the calling coroutine. The coroutine cannot be running a C function, a metamethod, or an iterator. Any arguments to `yield` are passed as extra results to resume.

14.9 debug - Debugging

This library provides the functionality of the debug interface to Agena programmes. You should exert care when using this library. The functions provided here should be used exclusively for debugging and similar tasks, such as profiling. Please resist the temptation to use them as a usual programming tool: they can be very slow. Moreover, several of its functions violate some assumptions about Agena code (e.g., that variables local to a function cannot be accessed from outside or that userdata metatables cannot be changed by Agena code) and therefore can compromise otherwise secure code.

All functions in this library are provided inside the `debug` table. All functions that operate over a thread have an optional first argument which is the thread to operate over. The default is always the current thread.

debug.debug ()

Enters an interactive mode with the user, running each string that the user enters. Using simple commands and other debug facilities, the user can inspect global and local variables, change their values, evaluate expressions, and so on. A line containing only the word `cont` finishes this function, so that the caller continues its execution.

Note that commands for **debug.debug** are not lexically nested within any function, and so have no direct access to local variables.

debug.funcname (level)

Returns the name of the function in which it (i.e. **debug.funcname**) has been called. The return is a string. It is a wrapper for "debug.getinfo(level, "n").name". By default, `level` 1 is used, but you may pass another `level`. If `level` is out of range, then **fail** will be returned. If the function name could not be determined, **null** will be returned. The function may be useful to create more flexible error messages.

debug.getconstants ()

Returns the internal set that stores global constants.

debug.getfenv (obj)

Returns the environment of object `obj`.

See also: **debug.setfenv**.

`debug.gethook ([thread])`

Returns the current hook settings of the thread, as three values: the current hook function, the current hook mask, and the current hook count (as set by the **debug.sethook** function).

`debug.getinfo ([thread,] function [, what])`

Returns a table with information about a function. You can give the function directly, or you can give a number as the value of `function`, which means the function running at level `function` of the call stack of the given thread: level 0 is the current function (**getinfo** itself); level 1 is the function that called **getinfo**; and so on. If `function` is a number larger than the number of active functions, then **getinfo** returns **null**.

The returned table may contain all the fields returned by **lua_getinfo**, with the string `what` describing which fields to fill in. The default for `what` is to get all information available, except the table of valid lines. If present, the option `'f'` adds a field named `func` with the function itself. If present, the option `'L'` adds a field named `activelines` with the table of valid lines. If present, the option `'g'` adds a field named `globals` with a table of variables that have been globally assigned. The `'a'` option adds a field called `arity` that includes the number of arguments - excluding possible varargs - expected by `function`. When given the `'v'` formatter, the 'varargs' entry indicates whether varargs can be passed to the function (**?** in its parameter list). `'V'` returns a table of all parameters and locally declared variables along with their current values. `'c'` returns activation record ar->i_ci setting.

For instance, the expression `debug.getinfo(1, 'n').name` returns a name of the current function, if a reasonable name can be found, and `debug.getinfo(print)` returns a table with all available information about the **print** function.

See also: **debug.funcname**.

`debug.getlocal ([thread,] level, local)`

This function returns the name and the value of the local variable with index `local` of the function at level `level` of the stack. (The first parameter or local variable has index 1, and so on, until the last active local variable.) The function returns **null** if there is no local variable with the given index, and raises an error when called with a `level` out of range. (You can call **debug.getinfo** to check whether the level is valid.)

Variable names starting with `'('` (open parentheses) represent internal variables (loop control variables, temporaries, and C function locals).

See also: **debug.getlocals**, **debug.setlocal**.

`debug.getlocals (level [, option])`

Like **debug.getlocal**, but returns a table of all local variables of a function running at level `level` of the stack. The array part of the table includes the variable names, the hash part the variables and their current values as key ~ value pairs. The other returns, in the following order are: number of local variables including procedure parameters, number of procedure parameters, a boolean indicating whether **?** (varargs) is part of the parameter list, a table with all unassigned local variables.

By passing any `option`, only an array of parameter and local variable names plus the number of parameters (first entries in the array) will be returned.

`debug.getmetatable (object)`

Returns the metatable of the given `object` or **null** if it does not have a metatable.

See also: **debug.setmetatable**.

`debug.getregistry ()`

Returns the registry table, see Chapter 6.31. Do not change values with integer keys - this would destroy occupied by userdata and could lead to undefined behaviour of the interpreter.

`debug.getrtable (f)`

Returns a reference to the internal remember table of procedure `f`. Opposed to **rtable.rget**, the function gives you direct read and write access to the remember table, so use it with care.

`debug.getstore (f)`

Returns a reference to the internal storage table of procedure `f`. You can both inspect this table as well as inject values into it. See **debug.setstore** or source file lib/mapm.agn or Chapter 6.25 for an example of how to store precomputed values, here Chebyshev coefficients, in a function, to be used later at function invocation.

`debug.getupvalue (f, up)`

This function returns the name and the value of the upvalue with index `up` of the function `f`. The function returns **null** if there is no upvalue with the given index.

See also: **debug.getupvalues**, **debug.nupvalues**, **debug.setupvalue**.

`debug.getupvalues (f)`

Returns all upvalues of an Agena closure `f` in a table, plus the number of upvalues. The first entry in the table depicts the first upvalue, and so on. If there are no upvalues, the return is **null** plus zero. The function does not accept closures written in C.

See also: **debug.getupvalue**, **debug.nupvalues**, **debug.setupvalue**.

`debug.nupvalues (f)`

Returns the number of upvalues in an Agena closure. The function does not accept closures written in C.

See also: **debug.getupvalue**, **debug.getupvalues**.

`debug.setfenv (object, t)`

Sets the environment of the given `object` to the given table `t`. Returns `object`.

See also: **debug.getfenv**.

`debug.sethook ([thread,] hook, mask [, count])`

Sets the given function as a hook. The string `mask` and the number `count` describe when the hook will be called. The string mask may have the following characters, with the given meaning:

- `'c'`**:** The hook is called every time Agena calls a function;
- `'r'`**:** The hook is called every time Agena returns from a function;
- `'l'`**:** The hook is called every time Agena enters a new line of code.

With a `count` different from zero, the hook is called after every `count` instructions.

When called without arguments, **debug.sethook** turns off the hook.

When the hook is called, its first parameter is a string describing the event that has triggered its call: `'call'`, `'return'` (or `'tail return'`), `'line'`, and `'count'`. For line events, the hook also gets the new line number as its second parameter. Inside a hook, you can call **getinfo** with level 2 to get more information about the running function (level 0 is the **getinfo** function, and level 1 is the hook function), unless the event is `'tail return'`. In this case, Agena is only simulating the return, and a call to **getinfo** will return invalid data.

`debug.setlocal ([thread,] level, local, value)`

This function assigns the value `value` to the local variable with index `local` of the function at level `level` of the stack. The function returns **null** if there is no local variable with the given index, and raises an error when called with a `level` out of range. (You can call **getinfo** to check whether the level is valid.) Otherwise, it returns the name of the local variable.

See also: **debug.getlocal**.

`debug.setmetatable (object, t)`

Sets the metatable for the given `object` to the given table `t` (which can be **null**).

See also: **debug.getmetatable**.

`debug.setstore (f, t)`

Sets all the key~value pairs in table `t` into the internal store of procedure `f`.

See also: **debug.getstore**.

`debug.setupvalue (f, up, value)`

This function assigns the value `value` to the upvalue with index `up` of the function `f`. The function returns **null** if there is no upvalue with the given index. Otherwise, it returns the name of the upvalue.

See also: **debug.getupvalue**.

`debug.system ()`

Returns a table with the following system information: The size of various C types (char, int, long, long long, float, double, int32_t, int64_t), the smallest and largest numeric values for C doubles, C long ints, C long long ints, and C unsigned long ints (all compiled into the Agena binary), the endianness of your platform, the hardware and the operating system for which the Agena executable has been compiled.

See also: **environ.kernel**.

`debug.traceback ([thread,] [message])`

Returns a string with a traceback of the call stack. An optional `message` string is appended at the beginning of the traceback. This function is typically used with **xpcall** to produce better error messages.

Chapter **Fifteen**

Graphics

15 Graphics

15.1 gdi - Graphic Device Interface package

As a *plus* package, this graphics interface is not part of the standard distribution and must be activated with the **import** statement, e.g. `import gdi`.

The gdi package provides functions to plot graphics either to a window or a PNG, GIF, JPEG, FIG, or PostScript file. It is available for the Solaris, Linux, Mac OS X for Intel CPUs, and Windows editions of Agena. There is an experimental OS/2 - ArcaOS version that is still experimental and it can create FIG and PostScript files only. The package is not available in the 64-bit editions.

The gdi package provides procedures to plot basic geometric objects such as points, lines, circles, ellipses, rectangles, etc.

It also provides means to easily plot graphs of univariate functions and geometric objects where the user does not need pay attention for proper axis ranges, mapping to the internal coordinate systems, etc.

15.1.1 Opening a File or Window

Operation starts by opening a device - window or file - with the **gdi.open** function. The function returns a device handle for later reference. Almost all functions provided by the package request this device handle.

```
> import gdi;
```

```
> d := gdi.open(640, 480);
```

15.1.2 Plotting Functions

Plot a point to the window at x=200 and y=100:

```
> gdi.point(d, 200, 100);
```

Plot a line between two points [200, 150] and [300, 200]:

```
> gdi.line(d, 200, 150, 300, 200);
```

Draw a circle and a filled circle. Besides giving the device number, pass a centre (x and y co-ordinates) and a radius.

```
> gdi.circle(d, 320, 240, 50);
```

```
> gdi.circlefilled(d, 400, 240, 50);
```

15.1.3 Colours, Part 1

All functions accept a colour option passed as an additional - the last - argument.

The colour must be given as an integer that must be determined by a call to the **gdi.ink** function. **gdi.ink** requires the device number, and three RGB colour values in the range [0 .. 1]. Each colour should be determined only once.

There are 26 predefined colours with numbers 0 to 25, automatically set at each invocation of a new device (call to the **gdi.open** function). Thus, these 26 basic colours do not need to be explicitly set with **gdi.ink**.

The default colours are:

0	white	7	light green	14	grey	21	purple
1	black	8	greenish	15	grey-blue	22	dark orange
2	blue	9	light sky-blue	16	bright green	23	purple
3	light blue	10	bordeaux	17	light greenish	24	light lilac
4	greenish	11	lilac	18	light sky-blue	25	yellow
5	cyan	12	light lilac	19	red		
6	sky-blue	13	khaki	20	purple		

```
> cyan := gdi.ink(d, .1, .5, .5);

> gdi.rectanglefilled(d, 200, 200, 400, 400, cyan);
```

If you want to set a default colour for all subsequent drawings, use **gdi.useink**.

15.1.4 Closing a File or Window

To finally close the window, use **gdi.close**.

```
> gdi.close(d);
```

15.1.5 Supported File Types

To create image files, simply pass the name of the file as the third argument to **gdi.open**. Agena determines the type of the image file from its suffix.

If a file name ends in `.png`, it creates a PNG file. If a file name ends in `.gif`, it creates a GIF file. If a file name ends in `.jpg`, it creates a JPEG file. Likewise, the suffix `.fig` creates a FIG, and `.ps` generates a PostScript file.

15.1.6 Plotting Graphs of Univariate Functions

The **gdi.plotfn** function plots graphs of functions in one real to a window or file. It accepts various options for colour, line thickness, line style, sizing, axis type, etc. The function takes care for opening a device, plotting the graph and axes, so that the user does not need to draw them manually. The function requires a function and the left and right border on the x-axis.

```
> import gdi alias

> plotfn(<< x -> x*sin(x) >>, -10, 10);
```

For further details and examples see **gdi.plotn**. For available plot options, see gdi.options. See **calc.nokspline** which along with **gdi.plotfn** generates a smoothed graph through a given list of interpolation points.

15.1.7 Plotting Geometric Objects Easily

Like **gdi.plotfn**, the gdi function **plot** outputs geometric objects in the Cartesian co-ordinate system with the point [0, 0] its centre. It accepts options for user-defined colours, window sizes, axis types, etc. The function opens a device automatically, plots all the objects that are stored in a PLOT data structure optionally along with axes, a user-defined background colour, etc.

The function requires the PLOT structure as the first argument, and any options as additional arguments. Contrary to **gdi.plotfn**, it does not accept left, right, lower or upper borders, for it determines the borders automatically.

A PLOT data structure is a sequence of the user-defined type 'PLOT', and contains the geometric objects with their positions and respective colours.

The following geometric objects can be drawn with **gdi.plot**:

Object	Name	Object	Name
arc	ARC	line	LINE
filled arc	ARCFILLED	point	POINT
circle	CIRCLE	rectangle	RECTANGLE
filled circle	CIRCLEFILLED	filled rectangle	RECTANGLEFILLED
ellipse	ELLIPSE	triangle	TRIANGLE
filled ellipse	ELLIPSEFILLED	filled triangle	TRIANGLEFILLED

A line stretching from [0, 0] to [1, 1] in grey colour (RGB values 0.5, 0.5, 0.5) for example is represented as follows:

LINE(0, 0, 1, 1, [0.5, 0.5, 0.5])

A PLOT structure can be created with the **gdi.structure** function that optionally accepts the minimum number of entries (for speed).

```
> import gdi alias;

> s := structure();
```

Any geometric objects is inserted into the structure with its respective **gdi.set*** function. The line LINE(0, 0, 1, 1, [0.5, 0.5, 0.5]) for example is added with the **gdi.setline** function:

```
> setline(s, 0, 0, 1, 1, [0.5, 0.5, 0.5]);
```

A PLOT structure can include any number of objects:

```
> setcircle(s, 0, 0, 0.5, [1, 0, 0]);
```

Finally, the **plot** statement puts them onto the screen:

```
> plot(s);
```

The following table shows the various functions to create objects:

Object	Function	Object	Function	Object	Function
arc	`setarc`	ellipse	`setellipse`	rectangle	`setrectangle`
filled arc	`setarcfilled`	filled ellipse	`setellipse-filled`	filled rectangle	`setrectangle-filled`
circle	`setcircle`	line	`setline`	triangle	`settriangle`
filled circle	`setcircle-filled`	point	`setpoint`	filled triangle	`settriangle-filled`

15.1.8 Colours, Part 2

The following colour names (of type string) are built in and are accepted by the **gdi.plot** and **gdi.plotfn** functions only, so that you must not define colours with **gdi.useink** or **gdi.ink** when plotting sets of points or graphs of functions:

```
'aquamarine', 'black', 'blue', 'bordeaux', 'brown', 'coral', 'cyan',
'darkblue', 'darkcyan', 'darkgrey', 'gold', 'green', 'grey', 'khaki',
'lightgrey','magenta', 'maroon', 'navy', 'orange', 'pink', 'plum', 'red',
'sienna', 'skyblue', 'tan', 'turquoise', 'violet', 'wheat', 'white',
'yellow', 'yellow2'.
```

15.1.9 GDI Functions

gdi.arc (d, x, y, r1, r2, a1, a2 [, colour])

Draws an arc around the centre [`x`, `y`] with x radius `r1`, y radius `r2`, and the starting and ending angles `a1`, `a2`, given in degrees [0 .. 360], on device `d`. A `colour` (an integer, see Chapter 15.1.3), may be given optionally.

`gdi.arcfilled (d, x, y, r1, r2, a1, a2 [, colour])`

Draws a filled arc around the centre [`x`, `y`] with x radius `r1`, y radius `r2`, and the starting and ending angles `a1`, `a2`, given in degrees [0 .. 360], on device `d`. The arc is filled with either the default colour, or the one given by `colour` (an integer, see Chapter 15.1.3).

`gdi.autoflush (d, state)`

Sets the auto flush mode for device `d` to either **true** or **false** (second argument). If `state` is **true** (the default), then after each graphical operation the output is flushed so that it is immediately displayed.

This may decrease performance significantly with a large number of graphical operations - Sun Sparcs seem to be the only exceptions -, so it is advised to

1. set `state` to **false** right after opening device `d` before calling any other function that plots something,
2. call **gdi.flush** after the graphical operations have been completed,
3. set `state` to **true** thereafter.

`gdi.background (d, c)`

Sets the background colour on device `d`. `c` must be a number determined by **gdi.ink**, see Chapter 15.1.3. Note that in Windows, the image is also cleared so that the background is properly displayed, whereas in UNIX, the image is not reset.

`gdi.circle (d, x, y, r [, colour])`

Draws a circle around the centre [`x`, `y`] with radius `r`, on device `d`. A `colour` (an integer, see Chapter 15.1.3), may be given optionally.

`gdi.circlefilled (d, x, y, r [, colour])`

Draws a filled circle around the centre [`x`, `y`] with radius `r`, on device `d`. The circle is filled with either the default colour, or the one given by `colour` (an integer, see Chapter 15.1.3).

`gdi.clearpalette (d)`

Removes all inks on device `d`.

`gdi.close (d)`

Closes the window or file referred to by device id `d`. If d points to a file, all image contents is saved to it.

`gdi.dash (d, s)`

Sets the line dash on device id `d`. The sequence `s` includes a vector of dash lengths (black, white, black, ...). If `s` is the empty sequence, a solid line is restored.

`gdi.ellipse (d, x, y, r1, r2 [, colour])`

Draws an ellipse around the centre [`x`, `y`] with x radius `r1`, and y radius `r2`, on device `d`. A `colour` (an integer, see Chapter 15.1.3), may be given optionally.

`gdi.ellipsefilled (d, x, y, r1, r2 [, colour])`

Draws a filled ellipse around the centre [`x`, `y`] with x radius `r1`, and y radius `r2`, on device `d`. The ellipse is filled with either the default colour, or the one given by `colour` (an integer, see Chapter 15.1.3).

`gdi.flush (d)`

Writes all buffered contents to the window or file referred to by device id `d`.

See also: **gdi.autoflush**.

`gdi.fontsize (d, s)`

Sets the font size `s` for text written by gdi.text, for device `d`.

See also: **gdi.text**.

`gdi.hasoption (t, o)`

Iterates a table `t` and returns true if one of its keys is equal to `o`.

See also: **gdi.options**.

`gdi.initpalette (d)`

Sets up basic colours on device `d`.

`gdi.ink (d, r, g, b)`

Returns a palette colour value - an integer - for the colour given by its RGB values `r` (red), `g` (green), and `b` (blue), for device `d`. `r`, `g`, and `b` must be numbers x with $0 \leq x \leq 1$. The palette colour value can be given as an optional argument in most of the **gdi** functions, or be used in the **gdi.useink** function. Subsequent calls with the same arguments return different palette values.

gdi.lastaccessed ()

Returns the id of the last accessed device as a number.

gdi.line (d, x1, y1, x2, y2 [, colour])

Draws a line from the first point [`x1`, `y1`] to the second point [`x2`, `y2`] on device `d`. A `colour`, an integer (see Chapter 15.1.3), may be given optionally.

gdi.lineplot (p [, options])
gdi.lineplot ([p1 [, p2, ···]], [, options])

Takes one or more tables or sequences consisting of points $x_k:y_k$ and generates a plot with all points connected by lines. x_k and y_k must be finite numbers. The function automatically determines the common proper borders automatically.

For more information see: **gdi.pointplot**, as **gdi.lineplot** is just a wrapper for the former with the 'connect' option set to **true**.

gdi.mouse (d [, offset])

Returns three numbers: the current horizontal and vertical positions of the mouse relative to the screen, and its button state *button_state*. The button state is coded as a positive integer.

By applying a bitmask to the button state, you can query whether the left or the right mouse button has been pressed:

- *button_state* && 0x0100 = 0x0100: left button has been pressed,
- *button_state* && 0x0400 = 0x0400: right button has been pressed.

gdi.open (width, height)

gdi.open (width, height, filename)

In the first form, opens a window with the given `width` and `height` and returns a device number (an integer) for later reference needed by all other **gdi** functions.

In the second form, creates the image file with name `filename`, the given `width` and `height` and returns a device number (an integer) for later reference needed by all other **gdi** functions.

The type of the image file format is determined by the suffix in `filename`:

Suffix	Resulting image file format	Example
.fig	FIG format	`'/export/home/misc/fern.fig'`
.gif	GIF format	`'c:/images/fractal.gif'`
.jpg	JPEG format	`'c:/images/fractal.jpg'`
.png	PNG format	`'c:/images/circle.png'`
.ps	PostScript format (DIN A4 size)	`'output.ps'`

`gdi.options (···)`

Checks the given plotting options for correctness and returns them in a new table, along with the defaults for options that have not been passed to this function. The function currently only works with the **gdi.plot**, **gdi.pointplot**, and **gdi.plotfn** functions.

Valid options (all key~value pairs) are:

Option (key)	Meaning (value)	Example
`'axes'`	`'none'` - do not print axes `'normal'` - print axes with labels and tick marks `'boxed'` - print axes at top and bottom, and at the left and the right side `'frame'` - print axes at the bottom and at the left side	`'axes':'normal'`
`'axescolour'`	defines the colour of the axes (a colour string, see Chapter 15.1.3)	`'axescolour':'red'`
`'bgcolour'`	sets the background colour (a colour string, see Chapter 15.1.3)	`'bgcolour':` `'yellow'`
`'colour'`	sets the default colour (a string, see Chapter 15.1.3) for the objects to be plotted. Note that the individual colour of an object overrides the one given by this option	`'colour':'navy'`
`'colourfn'`	sets a colouring function	`'colourfn':` `<< x -> ... >>`
`'file'`	indicates the name of the file (a string) to be created	`'file':'image.png'`
`'labels'`	if set to false, no labels are printed (default is **true**)	`'labels':false`
`'labelsize'`	sets the font size (a positive number) for axis labels (**gdi.plotfn** function only)	`'labelsize':6`
`'linestyle'`	sets the dash style (a positive number) for the graph to be plotted (**gdi.plotfn**, **gdi.lineplot**, and **gdi.pointplot** functions only)	`'linestyle':10`
`'maxtickmarks'`	sets the maximum number of tickmarks on both axes, by default is (around) 20.	`'maxtickmarks':5`
`'mouse'`	prints the current position of the mouse to the console. Click the right mouse button to finish. Default is **false**.	`'mouse':true`
`'res'`	resolution of the window or image file in pixels (pair of numbers)	`'res':(1024:768)`
`'square'`	in a plot, uses the same scale for the y-axis as given for the x-axis	`'square':true`

Option (key)	Meaning (value)	Example
`'thickness'`	sets the thickness (a positive number) of the line to be plotted (**gdi.plotfn**, **gdi.lineplot**, and **gdi.pointplot** functions only)	`'thickness':2`
`'title'`	sets the title (a string) for the plot (**gdi.plotfn** function only)	`'title':` `'Graph of sin(x)'`
`'titlecolour'`	sets the colour (a string, see Chapter 15.1.3) of the title (**gdi.plotfn** only)	`'titlecolour':` `'red'`
`'titlesize'`	sets the font size (a positive number) of the title (**gdi.plotfn** function only)	`'titlesize':15`
`'x'`	horizontal range (left and right border) over which the plot is displayed	`'x':(-2):2`
`'y'`	vertical range (lower and upper border) over which the plot is displayed	`'y':0:5`
`'xscale'`	sets the step size for the tick marks on the horizontal axis	`'xscale':0.5`
`'yscale'`	sets the step size for the tick marks on the vertical axis	`'yscale':0.5`

The function is written in Agena and included in the lib/gdi.agn file.

See also: **gdi.setoptions**.

gdi.plot (p [, options])

Plots PLOT structures stored in `p`. PLOT structures are points, lines, circles, triangles, rectangles, arcs, and ellipses, along with the information given by its optional INFO structure.

A PLOT structure is created by a call to **gdi.structure**, and the respective **gdi.set*** functions.

The function accepts all plot options (see **gdi.options**).

Example:

```
> p := gdi.structure();

> gdi.setline(p, 0, 0, 1, 1, 'navy');

> gdi.setcircle(p, 0, 0, 1, 'red');

> gdi.plot(p);

> gdi.plot(p, axes='normal', square=true, x=-2:2, y=-2:2);
```

The function is written in Agena and included in the lib/gdi.agn file.

```
gdi.plotfn (f, a, b [ [ c, d], options])
gdi.plotfn (ft, a, b [ [ c, d], options])
```

Plots graphs of one or more functions, with a straight line drawn between neighbouring points, which are automatically computed.

In the first form, the graph of the function `f` is plotted.

In the second form, by passing a table `ft` of functions, the graphs of the functions are plotted on one device - to one file or window.

If the `file` option is missing, the graphs are plotted in a window (UNIX/Mac and Windows, only). If the `file` option is given, the file type is determined by the suffix of the file you pass to this option.

`a` and `b` (both numbers with `a` < `b`) must be given explicitly and specify the horizontal range. If `c` and `d` are missing, the vertical range is determined automatically.

You may specify one or more options for proper layout of the graphs. See **gdi.options** for more details.

If a table of function is passed, you may specify an individual colour, line style, and the thickness for each of their graphs. Just pass a table of settings at the right-hand side of the respective option. See the examples below.

See **gdi.autoflush** if you experience performance problems while plotting.

Examples:

Plot the graph of the sine function on the horizontal range `a` to `b`. The vertical range is computed automatically.

```
> import gdi;

> gdi.plotfn(<< x -> sin(x) >>, -10, 10);
```

Plot the graph of the sine function on the horizontal range `a` to `b` and the vertical range `c` to `d`.

```
> gdi.plotfn(<< x -> sin(x) >>, -10, 10, -2, 2);
```

Specify a colour other than black:

```
> gdi.plotfn(<< x -> sin(x) >>, -10, 10, colour='red');
```

Give a specific thickness for the line:

```
> gdi.plotfn(<< x -> sin(x) >>, -10, 10, thickness=3);
```

Combine the options - their order does not matter:

```
> gdi.plotfn(<< x -> sin(x) >>, -10, 10, thickness=3, colour='red');
```

Plot two and more functions:

```
> gdi.plotfn([<< x -> sin(x) >>, << x -> cos(x) >>], -10, 10);
```

Give options, too:

```
> gdi.plotfn([<< x -> sin(x) >>, << x -> cos(x) >>], -10, 10,
>    colour='navy');
```

Specify individual colours. The graph of the sine function shall be red, the cosine function shall by cyan:


```
> gdi.plotfn([<< x -> sin(x) >>, << x -> cos(x) >>], -10, 10,
>    colour=['red', 'cyan']);
```

Choose another colour for the axes and another axes style:

```
> gdi.plotfn([<< x -> sin(x) >>, << x -> cos(x) >>], -10, 10,
>    colour=['red', 'cyan'], axescolour='grey', axes='boxed'
>    res=480:200);
```

Do not draw axes:

```
> gdi.plotfn([<< x -> sin(x) >>, << x -> cos(x) >>], -10, 10,
>    colour=['red', 'cyan'], axes='none');
```

If you want to set default options that will always be used by **plotfn** and that do not need to be specified with each call to **plotfn**, use **gdi.setoptions**:

```
> gdi.setoptions(colour='red', axescolour='grey');
```

```
> gdi.plotfn([<< x -> sin(x) >>, << x -> cos(x) >>], -10, 10)
```

The function is written in Agena and included in the lib/gdi.agn file.

See also: **calc.clampedspline**, **calc.nakspline**.

`gdi.point (d, x, y [, colour])`

Plots a point with co-ordinates [`x`, `y`] on device `d`. A `colour`, an integer (see Chapter 15.1.3), may be given optionally.

`gdi.pointplot (p [, options])`
`gdi.pointplot ([ p1 [, p2, ···] ], [, options])`

Takes one or more tables or sequences consisting of points x_k:y_k and generates a plot with no points connected by lines. x_k and y_k must be finite numbers. The function automatically determines the common proper borders automatically.

By passing the option `colour`=c, where c is either a string denoting a colour, or a table of strings denoting colours, you can set individual colours for the distributions. The default is `'black'`.

By passing the option `symbol`=s, where s is the name of a symbol or a table of strings denoting symbols, each point in a distribution is plotted accordingly. Supported symbols are: `'cross'`, `'circle'`, `'circlefilled'`, `'box'`, `'boxfilled'`, `'triangle'`, `'trianglefilled'`, `'crosscircle'`, and `'dot'`. The default is `'dot'`.

The size of the symbols can be controlled by the `symbolsize` option which denotes a radius in pixels. Only one common size can be set for all distributions passed. The default is 3.

Alternatively, by passing the `connect`=**true** option, you can connect all points in each distribution with a line.

The function supports various plotting options, see **gdi.options**.

In the first form, only one distribution `p` is passed, in the second form you can pass various distributions `p1`, `p2`, etc. by putting them into a table.

The function ignores y-values if they evaluate to **infinity** or **undefined**.
Example:

```
> s := seq(0.1, 0.2, 0.1, 0.3, 1, 2, 5, -1, 0);

> p := sequences.new( << x -> x:s[x] >>, 1, size s);

> s1 := << x -> ln(x) >> @ s;

> p1 := sequences.new( << x -> x:s1[x] >>, 1, size s1);

> gdi.pointplot([p, p1], colour=['red', 'black'],
>    symbol=['circle', 'cross'], symbolsize=5, connect=true);
```

The function is written in Agena and included in the lib/gdi.agn file.

See also: **gdi.lineplot**.

gdi.rectangle (d, x1, y1, x2, y2 [, colour])

Draws a rectangle with the lower left and upper right corners [x1, y1] and [x2, y2] on device d. A `colour` (an integer, see Chapter 15.1.3), may be given optionally for the lines.

gdi.rectanglefilled (d, x1, y1, x2, y2 [, colour])

Draws a filled rectangle with the lower left and upper right corners [x1, y1] and [x2, y2] on device d. The rectangle is filled with either the default colour, or the one given by `colour` (an integer, see Chapter 15.1.3).

gdi.reset (d)

Clears the entire window or image file contents of device d.

gdi.resetpalette (d)

Clears the colour palette by removing all inks and reallocates basic colours, on device d.

gdi.setarc (s, x, y, r1, r2, a1, a2 [, colour [, thickness]])

Inserts an arc around the centre [x, y] with x radius r1, y radius r2, and the starting and ending angles a1, a2, given in degrees [0 .. 360], to PLOT structure s. The optional `colour` argument may be either a string denoting a colour like 'black', 'red', etc., or a table with three RGB numeric values in the range 0 .. 1. `thickness` is the thickness of the arc, with 1 its default.

gdi.setarcfilled (s, x, y, r1, r2, a1, a2 [, colour])

Inserts a filled arc around the centre [x, y] with x radius r1, y radius r2, and the starting and ending angles a1, a2, given in degrees [0 .. 360], to PLOT structure s. The optional `colour` argument may be either a string denoting a colour like 'black', 'red', etc., or a table with three RGB numeric values in the range 0 .. 1.

gdi.setcircle (s, x, y, r [, colour [, thickness]])

Inserts a circle around the centre [x, y] with radius r, to PLOT structure s. The optional `colour` argument may be either a string denoting a colour like 'black', 'red', etc., or a table with three RGB numeric values in the range 0 .. 1. `thickness` is the thickness of the circle, with 1 its default.

gdi.setcirclefilled (s, x, y, r [, colour])

Inserts a filled circle around the centre [x, y] with radius r, to PLOT structure s. The optional `colour` argument may be either a string denoting a colour like 'black', 'red', etc., or a table with three RGB numeric values in the range 0 .. 1.

`gdi.setellipse (s, x, y, r1, r2 [, colour [, thickness]])`

Inserts an ellipse around the centre [`x`, `y`] with x radius `r1`, and y radius `r2`, to PLOT structure `s`. The optional `colour` argument may be either a string denoting a colour like `'black'`, `'red'`, etc., or a table with three RGB numeric values in the range 0 .. 1. `thickness` is the thickness of the ellipse, with 1 its default.

`gdi.setellipsefilled (s, x, y, r1, r2 [, colour])`

Inserts a filled ellipse around the centre [`x`, `y`] with x radius `r1`, and y radius `r2`, to PLOT structure `s`. The optional `colour` argument may be either a string denoting a colour like `'black'`, `'red'`, etc., or a table with three RGB numeric values in the range 0 .. 1.

`gdi.setinfo (s, ···)`

Inserts information on the minimum and maximum values (x- and y values) and their scaling of all the geometric objects included in the PLOT data structure `s` into its INFO substructure. The INFO object always is the last element in `s`.

The options `xdim=a:b` and `ydim=c:d` set the x-range and y-range on which objects will be plotted, respectively, where `a`, `b`, `c`, `d` are numbers (i.e. borders). The `square = true` option scales the x and y dimensions equally, the `square = false` does not.

The information is useful so that **gdi.plot** can automatically determine the proper plotting ranges for `s`.

Example:

```
> gdi.setinfo(s, xdim = 0:10, ydim = -5:5, square = false);
```

`gdi.setline (s, x1, y1, x2, y2 [, colour [, thickness]])`

Inserts a line drawn from point (`x1`, `y1)` to point (`x2`, `y2`) with the optional `colour` into the PLOT structure `s`. `x1`, `y1`, `x2`, `y2` should be numbers. `colour` may be either a string denoting a colour like `'black'`, `'red'`, etc., or a table with three RGB numeric values in the range 0 .. 1. `thickness` is the thickness of the line, with 1 its default.

`gdi.setoptions (···)`

Checks the given plotting options (all key~value pairs) for correctness and sets them as the respective defaults for subsequent calls to the **gdi.plot** and **gdi.plotfn** functions.

For a list of valid plotting options, see **gdi.options**.

Internally, the function assigns the given options to the global environment variable **environ.gdidefaultoptions** which is checked by **gdi.plot** and **gdi.plotfn**.

gdi.setpoint (s, x, y [, colour])

Inserts a point with co-ordinates [x, y] to PLOT structure s. The optional `colour` argument may be either a string denoting a colour like `'black'`, `'red'`, etc., or a table with three RGB numeric values in the range 0 .. 1.

gdi.setrectangle (s, x1, y1, x2, y2 [, colour [, thickness]])

Inserts a rectangle with the lower left and upper right corners [x1, y1] and [x2, y2] to PLOT structure s. The optional `colour` argument may be either a string denoting a colour like `'black'`, `'red'`, etc., or a table with three RGB numeric values in the range 0 .. 1. `thickness` is the thickness of the arc, with 1 its default.

gdi.setrectanglefilled (s, x1, y1, x2, y2 [, colour])

Inserts a filled rectangle with the lower left and upper right corners [x1, y1] and [x2, y2] to PLOT structure s. The optional `colour` argument may be either a string denoting a colour like `'black'`, `'red'`, etc., or a table with three RGB numeric values in the range 0 .. 1.

gdi.settriangle (s, x1, y1, x2, y2, x3, y3 [, colour [, thickness]])

Inserts a triangle with the corners [x1, y1], [x2, y2], and [x3, y3] to PLOT structure s. The optional `colour` argument may be either a string denoting a colour like `'black'`, `'red'`, etc., or a table with three RGB numeric values in the range 0 .. 1. `thickness` is the thickness of the arc, with 1 its default.

gdi.settrianglefilled (s, x1, y1, x2, y2, x3, y3 [, colour])

Inserts a filled triangle with the corners [x1, y1], [x2, y2], and [x3, y3] to PLOT structure s. The optional `colour` argument may be either a string denoting a colour like `'black'`, `'red'`, etc., or a table with three RGB numeric values in the range 0 .. 1.

gdi.structure ([n])

Creates a PLOT data structure with n pre-allocated entries. Of course, the structure may contain less or more entries. If n is not given, no pre-allocation is done which may slow down inserting new objects into s later in a session. The return is the PLOT data structure (a sequence of user type 'PLOT').

See also: **gdi.setinfo**.

gdi.system (d, x, y, xs, ys)

Sets the user's co-ordinate system on device d, where x, y, xs, and ys are numbers. The pixel [x, y] determines the origin. The horizontal unit is given in xs pixels, the vertical unit in ys pixels. The function returns nothing.

```
> d := open(640, 480);

> gdi.system(d, 320, 240, 320, 240);

> gdi.line(d, -1, 0, 1, 0);

> gdi.line(d, 0, -1, 0, 1);
```

gdi.text (d, x, y, str [, colour])

Prints the string `str` at [`x`, `y`] on device `d`. A text `colour` (an integer), may be given optionally.

See also: **gdi.fontsize**.

gdi.thickness (d, t)

Sets the default thickness for all lines to `t` pixels, on device `d`.

gdi.triangle (d, x1, y1, x2, y2, x3, y3 [, colour [, thickness]])

Draws a triangle with the corners [`x1`, `y1`], [`x2`, `y2`], and [`x3`, `y3`] on device `d`. A `colour` (an integer, see Chapter 15.1.3), may be given optionally for the lines. `thickness` is the thickness of the triangle, with 1 its default.

gdi.trianglefilled (d, x1, y1, x2, y2, x3, y3 [, colour])

Draws a filled triangle with the corners [`x1`, `y1`], [`x2`, `y2`], and [`x3`, `y3`] on device `d`. The triangle is filled with either the default colour, or the one given by `colour` (an integer, see Chapter 15.1.3).

gdi.useink (d, c)

Sets the default colour `c` (a number) for all subsequent drawings, on device `d`. `c` must be a number determined by **gdi.ink**.

15.2 fractals - Library to Create Fractals

As a *plus* package, in Solaris, Linux, Mac OS X, and Windows, this library is not part of the standard distribution and must be activated with the **import** statement, e.g. `import fractals`.

Since it needs **gdi** graphics functions, it is of no use in DOS. The OS/2 - ArcaOS version is still experimental. The package is not available in the 64-bit edition.

The library creates fractals and includes three types of functions:

1. escape-time iteration functions like **fractals.mandel**,
2. auxiliary mathematical functions lie **fractals.flip**,
3. fractals.draw to draw fractals using escape-time iteration functions.

See Chapter 15.2.3 for some examples.

15.2.1 Escape-time Iteration Functions

fractals.amarkmandel (x, y, iter, radius)

This function computes the escape-time fractal created by Mark Peterson of the formula:

$$z := z^2 * c^{0.1} + c$$

It returns the number of iterations a point [`x`, `y`] needs to escape `radius`. The maximum number of iterations conducted is given by `iter`.

See also: **fractals.markmandel**.

fractals.albea (x, y, iter, radius)

This function calculates the Julia set of the formula lambda * bea(z), where lambda is the point 1!0.4 and z =`x`!`y`, and `iter` is the maximum number of iteration. Its return is the number of iterations the function needs to escape `radius`. The function is written in Agena (see lib/fractals.agn).

See also: **fractals.lbea**.

fractals.alcos (x, y, iter, radius)

This function calculates the Julia set of the formula lambda * cos(z), where lambda is the point 1!0.4 and z =`x`!`y`, and `iter` is the maximum number of iteration. Its return is the number of iterations the function needs to escape `radius`. The function is written in Agena (see lib/fractals.agn).

`fractals.alcosxx (x, y, iter, radius)`

This function calculates the Julia set of the formula lambda * cosxx(z), where lambda is the point 1!0.4 and z =$x!y$, and `iter` is the maximum number of iteration. Its return is the number of iterations the function needs to escape `radius`. The function is written in Agena (see lib/fractals.agn).

The function implements FRACTINT's buggy cos function till v16, and creates beautiful fractals.

`fractals.alsin (x, y, iter, radius)`

This function calculates the Julia set of the formula lambda * sin(z), where lambda is the point 1!0.4 and z =$x!y$, and `iter` is the maximum number of iteration. Its return is the number of iterations the function needs to escape `radius`. The function is written in Agena (see lib/fractals.agn).

`fractals.anewton (x, y, iter, radius)`

This function implements Newton's formula for finding the roots of z^3 - 1, with z = $x!y$, and returns the number of iterations it takes for an orbit to be captured by a root. The iteration formula itself is

$$z := z - (z^3-1)/(3*z^2)$$

The function stops if $|z^3-1|$ < `radius` or the maximum number of iterations `iter` is reached. The function is written in Agena (see lib/fractals.agn).

See also: **fractals.newton**.

`fractals.lbea (x, y, iter, radius)`

This function calculates the Julia set of the formula lambda * bea(z), where lambda is the point 1!0.4 and z =$x!y$, and `iter` is the maximum number of iteration. Its return is the number of iterations the function needs to escape `radius`. The function is implemented in C.

See also: **fractals.albea**.

`fractals.mandel (x, y, iter, radius)`

This function computes the Mandelbrot set of the formula

$$z := z^2 + c$$

using complex arithmetic. It returns the number of iterations a point [x, y] needs to escape `radius`. The maximum number of iterations conducted is given by `iter`. The function is implemented in C.

`fractals.mandelbrot (x, y, iter, radius)`

Like **fractals.mandel**, but written in Agena and using complex arithmetic.

`fractals.mandelbrotfast (x, y, iter, radius)`

Like **fractals.mandel**, but written in Agena and using real arithmetic.

`fractals.mandelbrottrig (x, y, iter, radius)`

Like **fractals.mandel**, but written in Agena and using real arithmetic and trigonometric functions (see lib/fractals.agn).

`fractals.markmandel (x, y, iter, radius)`

Like **fractals.amarkmandel**, but implemented in C.

`fractals.newton (x, y, iter, radius)`

Like **fractals.anewton**, but implemented in C.

15.2.2 The Drawing Function fractals.draw

The function takes an escape-time iterator, various other parameters, and creates either image files or windows of fractals. By default a window is opened (see file option on how to create image files).

`fractals.draw (iterator, x_centre, y_centre, x_width [, options])`

Draws a fractal given by the escape-time iterator function `iterator` with image centre [`x_centre`, `y_centre`] and of the total length on the x-axis `x_width`. `x_centre` and `y_centre` are numbers whereas `x_width` is a positive number.

Options are:

Option	Meaning	Example
colour ~ f	a colouring function `f` of the form f := << x -> r, g, b >>. Predefined functions are: red, blue, violet, cyan, cyannew.	colour ~ << x -> 0, 0, 0.05*x >> colour ~ blue
file~'filename.suf'	creates a GIF, PNG, or JPEG file, if the file suffix is .gif, .png, or .jpg	file ~ 'mandel.gif'
iter ~ n	maximum number of iterations with `n` a positive number; default is 128	iter ~ 512
lambda ~ p	lambda value `p`, a complex number, for fractals.[a]l* functions like **albea**	lambda ~ 1!0.4

Option	Meaning	Example
map ~ 'filename.map'	FRACTINT colour map to be used to draw the fractal. The FRACTINT maps can be downloaded separately from: http://agena.sourceforge.net/downloads.html#fractintmaps Put these files into the share folder of your Agena distribution, preserving the subfolder fractint. A valid path may thus be: /usr/agena/share/fractint. Alternatively, set the environment variable environ.fractintcolourmaps to the folder where your map files reside.	map ~ 'basic.map'
mouse ~ bool	display pointer co-ordinates on console after image has been finished, if `bool` = **true**. Default: `bool` = **false**. Click the right mouse button to quit printing co-ordinates.	mouse ~ true
radius ~ r	iteration radius `r`, a positive number	radius ~ 2
res ~ width:height	resolution of the window or image, with `width` and `height` positive numbers. Default is 640:480	res ~ 1024:768
update ~ n	with `n` a non-negative number: determines the number of rows after an image is being flushed to a file or window during computation	

Notes on the **update** option:

On all operating systems the default is 1. This behaviour can globally be changed in a session by assigning a non-negative integer to the environment variable **environ.fractscreenupdates**.

In Sun x86 Solaris and Linux, update ~ 0 is the fastest, but when outputting to a window, it does not plot anything while the fractal is being computed (of course, if computation finishes, the fractal will be displayed).

Sparcs do not show any effect when changing the update rate, at least with XVR-1200 VGAs. The same applies to Microsoft Windows XP and 7, as well as Mac OS X 10.5.

15.2.3 Examples

```
> import fractals alias

> draw(fractals.lbea, 1.75, 0.5, 0.001, map='grayish.map', radius=1024,
>     iter=1024, lambda=1!0.4);
```


There are further examples at the bottom of the lib/fractals.agn file residing in the main Agena library folder.

```
> draw(mandel, -1.0037855135, 0.2770816775, 0.086686273, iter=255);

> draw(mandel, -1.0037855135, 0.2770816775, 0.086686273, file='out.png',
> iter=255, res=1024:768);  # create a PNG file of the Mandelbrot set
```

Chapter **Sixteen**

Utilities

16 Utilities

16.1 utils - Utilities

The **utils** package provides miscellaneous functions.

utils.calendar ([x])

Converts `x` seconds (an integer) elapsed since the beginning of an epoch to a table representing the respective calendar date in your local time. The table contains the following keys with the corresponding values:

- 'year' (integer)
- 'month' (integer)
- 'day' (integer)
- 'hour' (integer)
- 'min' (integer)
- 'sec' (integer)
- 'wday' (integer, day of the week)
- 'yday' (integer, day of the year)
- 'DST' (Boolean, is Daylight Saving Time)

If `x` is **null** or not specified, then the current system time will be returned. If `x` is invalid, the function will issue **fail**.

See also: **os.now**.

utils.checkdate (obj)

utils.checkdate (year, month, day [, hour [, minute [, second]]])

In the first form, receives a date of the form year, month, date [, hour [, minute [, second]]], with these values in table or sequence `obj` being integers, and checks whether the given date and optionally time exists and returns **true** or **false**.

In the second form, receives the given integers, and conducts the same operation.

utils.decodea85 (str)

Decodes the ASCII85 encoded string `str` and returns the result as a string.

See also: **utils.encodea85**.

utils.decodeb64 (str)

Decodes the Base64 encoded string `str` and returns the result as a string.

See also: **utils.encodeb64**.

`utils.decodeb85 (str [, option])`

Decodes the Base85 Z85 encoded string `str` and returns the result as a string. If `option` is set to **true**, then the function determines which characters in `str` might be invalid. The default is **false**.

See also: **utils.encodeb85**.

`utils.decodexml (str [, options])`

Reads a string `str` containing an XML stream and converts it into a dictionary.

You can pass one or two options in any order:

If the Boolean option **false** is given, the function does not automatically try to convert strings representing numbers, complex numbers and the Booleans **true**, **false**, and **fail** into the proper Agena representation.

If the option `'nocomment'` is given, the function does not return XML comments.

The function provides some checking (basic syntax and balanced tags), and supports namespaces, XML and DOCTYPE declarations, comments and processing instructions. If a XML tag includes hyphens or colons, then they are converted to underscores in the corresponding Agena dictionary key.

Since the function does not return processing instructions, you may want to have a look at the auxiliary utils.aux.decoderawxml function included in the lib/library.agn file which returns a user-defined table containing processing instructions in the `xarg` tag.

The function is written in Agena and included in the `libary.agn` file.

Here is an example:

```
> xmlstr := '<?xml version="1.0"?>
> <Data>
>    <Name1>Agena</Name1>
>    <Name2>1</Name2>
>    <Name3>1.1</Name3>
>    <Name4>1.1+2.2*I</Name4>
> </Data>
> <Lang:Info-All>
>    <Name action="interpret">Agena</Name>
>    <Version>1.6.1</Version>
> </Lang:Info-All>
> <!-- this is a comment -->
> <Motto>The Power of Procedural Programming</Motto>'

> utils.decodexml(xmlstr):
[Data ~ [Name1 ~ Agena, Name2 ~ 1, Name3 ~ 1.1, Name4 ~ 1.1+2.2*I],
Lang_Info_All ~ [Name ~ Agena, Version ~ 1.6.1], Motto ~ The Power of
Procedural Programming, header ~ <?xml version="1.0"?>]
```

```
> for i, j in ans do print(i, j) od
Lang_Info_All    [Name ~ Agena, Version ~ 1.6.1]
Motto   The Power of Procedural Programming
Data    [Name1 ~ Agena, Name2 ~ 1, Name3 ~ 1.1, Name4 ~ 1.1+2.2*I]
header  <?xml version="1.0"?>
```

The function is quite slow when parsing deeply nested XML structures, but it is more exact than **xml.decodexml**. If you need to parse only certain portions of an XML stream, just extract them from the string using the **strings.match** function before applying utils.decodexml.

See also: **utils.encodexml**, **utils.readxml**.

utils.encodea85 (str)

Encodes a string `str` into ASCII85 format and returns it as a string.

See also: **utils.decodea85**.

utils.encodeb64 (str)

Encodes a string `str` into Base64 format and returns it as a string.

See also: **utils.decodeb64**.

utils.encodeb85 (str)

Encodes a string `str` into Base85 Z85 format and returns it as a string.

See also: **utils.decodeb85**.

utils.encodexml (obj [, indent [, flag]])

Encodes a dictionary `obj` of the same format as created by **utils.readxml** into XML format.

If `indent` (a non-negative number) is not given the number of white space indentations is 3.

If any value is given for `flag`, the return is a flat table of substrings, else the return is one concatenated string.

See also: **utils.decodexml**.

```
utils.findfiles (d, what [, options])
utils.findfiles (obj, what [, options])
```

Searches a single file - or searches a directory for all the files - that include a certain string or which satisfy a given condition.

In the first form, the directory to be searched is denoted by the first argument `d`, a string, which may include file wildcards. `d` may also denote a single file. In the second form, `obj` is a table of a table with file names of type string, and the absolute path to the directory containing the given files. (**os.list** returns such a table.)

The second argument `what` can either be a string to be searched for, or a procedure of one argument that describes a satisfying condition and which should result in either **true** or **false**.

The returns are two lists: the first list includes all the names of the files where the search has been successful, and the second lists includes all files that could not be read due to errors, for example because of missing read permissions.

By default, the function searches all files line by line for a given search criterion. Pass the option `'whole'` if the search criterion should be applied to the entire file, i.e. to search in the string concatenation of all the lines of a file, so that line breaks do not matter.

By passing the further option `'r'`, the function also searches recursively in all respective subfolders.

Options may be given in any order after the second argument what.

Examples:

```
> utils.findfiles('*.c', '#define'):

> utils.findfiles('*.c', << x -> '#define' in x = 1 >>, 'whole'):

> utils.findfiles([['a.txt', 'b.txt'], 'c:/text'], 'hello'):
```

See also: **io.infile**, **io.readfile**.


```
utils.hexlify (str)
```

Converts a string `str` to its hexadecimal representation and returns a new string where each character in `str` is replaced by a two-digit hexadecimal value. The resulting string is twice as long as `str`.

See also: **utils.unhexlify**.

utils.posrelat (pos, len)

If `pos` represents a negative integer index, returns the respective positive index for the given number `len` of items in a structure, otherwise returns `pos`. The function is written in the Agena language and included in the lib/library.file.

utils.readcsv (filename [, options [, fn]])

Reads a comma-separated value (CSV) file and returns its contents in a sequence. The delimiter of the fields in a line by default is a semicolon.

If a line contains more than one field, then the respective fields are returned in a sequence[29]. If a line contains only one field, then it will be returned without including it in a sequence[30]. If a line contains nothing, i.e. '\n', it is by default ignored[31].

Strings containing numbers are automatically converted to numbers.

Options can be passed as pairs:

Left pair element	Right pair element	Example
convert	**true** or **false**: If **false**, do not attempt to convert strings to numbers. Default: **true**.	`convert = true`
comma	**true** or **false**: If a field contains a string recognised as a number by **strings.iscenumeric** - i.e. with a decimal comma instead of a decimal dot - this option automatically transforms the value to an Agena number if the option evaluates to **true**. Default is **false**. This option is applied before checking for the `convert` option.	`comma = true`
delim	A string. Use this string as the delimiter instead of a semicolon which is the default.	`delim = '|'`
dictionary	Returns a dictionary instead of a sequence with the dictionary keys defined by the values in the row passed with this new option, where the row can be depicted by a field number or field id (a string). The values in the `key` row should be unique.	`dictionary = 1` `dictionary = 'ID'`

---

[29] See the `flat` option to override this behaviour.

[30] See the `newseq` option to override this behaviour.

[31] See the `skipemptylines` option to override this behaviour.

Left pair element	Right pair element	Example
field	a positive integer: If given, only the given field in the CSV file is extracted, else all fields are returned.	`field = 3`

fields	A table or sequence of positive integers. If given, only the fields given in this table or sequence are returned, and in the order of the elements in this table or sequence; if not given, all fields are returned.
	
	If a CSV file contains a header, then column numbers or strings denoting the field name can be passed, and column numbers and field names can be mixed.
		`fields = [3, 1, 5]`
		
		`fields =`
		`   ['name', 'phone']`
		
		`fields =`
		`   ['name', 2]`

flat	**true** or **false**: If **true**, do not return values in each line in a new sequence. Default: **false**.	`flat = true`
header	**true** or **false**: If **true**, ignore the very first line. Default: **false**.	`header = true`

ignore	a procedure returning either **true**, **false**, or **fail**. If given, the procedure is applied to each line of the CSV file and if it evaluates to **true**, it does not process the line and proceeds with the next one.	`ignore =`
		`   << x ->`
		`     'text' in x`
		`     <> null`
		`   >>`

ignorespaces	**true** or **false**: all spaces in a line are deleted before returning the fields. Default is **false**.	`ignorespaces = true`

mapfields	A table or sequence of pairs of the form posint:procedure. Applies the given function to a specific field in the CSV file.
	
	If a CSV file contains a header, then column numbers or strings denoting the field name can be passed along with the procedures, and column numbers and field names can be mixed.
		`mapfields =`
		`   [1:f, 3:g]`
		
		`mapfields =`
		`   ['name':f, 2:g]`

Left pair element	Right pair element	Example
newseq	**true** or **false**: if only one field, i.e. one value per line, is stored in the CSV file, always put this single value in each line into a new sequence (**true**), resulting in a sequence of sequences returned by **readcsv**; otherwise simply add it to the flat sequence returned by the function, which is the default (**false**).	`newseq = true`
output	A string. If the right-hand side is `'record'`, then a dictionary will be returned, with its keys being defined by the tokens in the first line of the file (if the header=**true** option is also given), otherwise a table array will be returned.	`output = 'record'`
remove	`'quotes'` or `'doublequotes'`, or both. If `'quotes'` is given, enclosing single quotes are removed from the CSV field. If `'doublequotes'` is given, enclosing double quotes are removed from the CSV field (the default, see removedoublequotes option to prevent this). You cannot remove *both* single *and* double quotes.	`remove = 'quotes'`
remove-doublequotes	If set to **true**, removes enclosing double quotes from a field if present (the default). If set to **false**, enclosing double quotes are not deleted.	`removedoublequotes = false`
skipemptylines	**true** or **false**: If **true**, do not return empty lines. Default is **true**.	`skipemptylines = true`
skipfaultylines	**true** or **false**; if set to **true**, ignores all lines in a CSV file that do not have the same number of fields as there are in the CSV header, or the first CSV line if a header is non-existent. Default is **false**. If set to **true**, the function also returns all skipped lines as a second result.	`skipfaultylines = true`
skipspaces	**true** or **false**: If **true**, do not return lines consisting of spaces only. Default is **false**.	`skipspaces = true`

Left pair element	Right pair element	Example

subs	a pair, or a table or sequence of pairs x:y. For each line read from the CSV file, replaces x with y. If you pass a function as the last argument, substitution is done before finally mapping this function on the return.	`subs = '':undefined`
		`subs = ['':undefined,`
		` 'HUGE_VAL':infinity]`

You may also optionally pass a function `fn` - at any position in the argument list - to be mapped on each value of the input to be returned, or mix options given as pairs and a function to be applied to each value to be returned, e.g.:

```
> L := utils.readcsv('data.dat', delim=' ', flat=true, << x -> x^2 >>);
```

The function is written in Agena and included in the `libary.agn` file.

See also: **columns**, **descend**, **io.lines**, **io.readlines**, **utils.readxml**, **utils.writecsv**, **skycrane.readcsv**.

utils.readini (filename [, options])

Reads a traditional initialisation file and returns its contents as a table. Initialisation files supported look like the following:

```
#
# This is an example of an ini file
#
; Pizzas

Taxi=Pizza Cab
Agena=

[Pizza] ; <- this is a section name

Ham       = yes;  <- and this is a key~value pair
Mushrooms = true ;
Capres    = 0
Cheese    = "Non" ;
Price = 3.99
Preis=3,99
```

A line beginning with a hash (#), followed optionally by one or more characters, is completely ignored.

In a line, any text starting with a semicolon is also skipped. Key~value pairs may be separated by one or more white spaces.

The result is a table.

The file is parsed from top to bottom. As long as no section name has been given (here `[Pizza]`), any key~value pairs encountered are entered into the table as such.

If a section name is given, then a subtable of the form section ~ [key ~ value pairs] is stored to the resulting main table.

If a key is given, but now value, then the corresponding value will be the empty string. Values may also be enclosed in double quotes, but double quotes will be stripped of during import.

By default, any number values are automatically transformed to numbers, and the strings `'true'`, `'false'`, or `'fail'` are converted to Booleans, and all other values are returned as strings. You may prevent any conversion by passing the `convert=false` option.

If the option `comma=true` is given, then all floating point values containing a decimal comma are converted to a representation with a decimal dot. Default is `comma=false`.

The option `sections=true` reads only the section names in the ini file and returns them in the order of occurrence in a table array. Default is `sections=false`.
The results of reading the above ini file will look as follows if no option is given:

```
[Agena ~ , Taxi ~ 'Pizza Cab', Pizza ~ [Capres ~ 0, Cheese ~ Non, Ham ~
yes, Mushrooms ~ true, Preis ~ 3,99, Price ~ 3.99]]
```

See also: **utils.writeini**.

utils.readxml (filename [, options])

Reads an XML file and returns its data in an Agena dictionary.

You can pass one or two options in any order:

If the Boolean option **false** is given, the function does not automatically try to convert strings representing numbers, complex numbers and the Booleans **true**, **false**, and **fail** into the proper Agena representation.

If the option `'nocomment'` is given, the function does not return XML comments.

For further information on how the function works, see **utils.decodexml**.

See also: **utils.decodexml**, **utils.readcsv**, **xml.readxml**.

`utils.singlesubs (str, sp)`

Substitutes individual characters in string `str` by corresponding replacements in sequence `sp`. The return is a new string. Note that the function tries to find a replacement for a single character in `str` by determining its integer ASCII value n and then accessing index n in `sp`. If an entry is found for index n, then the character is replaced, otherwise the character remains unchanged.

For an example, check the **strings.diamap** procedure in the lib/library.agn file.

`utils.speed (n, f [, ···])`

Receives a positive integer `n`, a function `f`, and any optional arguments, and executes the function `n` times. The function returns the execution time in seconds.

If you want to check the speed of an operator, you have to enclose it in a function, e.g.:

```
> utils.speed(1k, << x -> sin x >>, 0):
```

See also: **time**.

`utils.unhexlify (str)`

Does the opposite of **utils.hexlify**.

`utils.uuid ([x])`

Creates a random version 4 universally unique identifier (UUID) by exclusively producing random numbers, and returns a string of 32 characters. If its argument `x` is **null**, the nil UUID will be returned (i.e. the template), otherwise, `x` has no effect.

See also: **environ.ref**, **factory.count**, **math.random**, **sema.open**.

`utils.writecsv (obj, filename [, options])`

Creates a comma-separated value (CSV) file. The function writes all values or keys and value(s) of a table, set, sequence or register `obj` to a text file given by `filename`. If `obj` includes a structure, then each element of the respective structure is written on the same line. Otherwise, each value or key ~ value pair is written on a separate line.

By default only values are written, the keys are ignored, but check the 'key' option below.

The following options can be passed as pairs:

Left pair element	Right pair element	Example
delim	A string. Use this string as the delimiter instead of a semicolon which is the default.	`delim = '|'`
dot	A single character of type string. With numbers, a decimal dot is replaces with the given character. Default: no replacement.	`dot = ','`
enclose	A string. Each value to be written is enclosed with this string.	`enclose = '\"'`
header	A string written to the very first line. Default: no header is written.	`header = 'A;B;C'`
key	A Boolean. If **true**, writes the respective index of the structure at the beginning of each line. Default: **false**, i.e. indices are not written.	`key = true`

The function returns nothing, is written in Agena and included in the lib/library.agn file.

Example:

```
> obj := seq(seq(1.1, 2, 3), seq(4, 5.1, 6), seq(7, 8, 9));
```

```
> utils.writecsv(obj, 'c:/out.csv', delim='|', dot=',');
```

creating a file with the contents:

```
1|1,1|2|3
2|4|5,1|6
3|7|8|9
```

See also: **utils.readcsv**, **skycrane.readcsv**.

utils.writeini (obj, filename [, options])

Creates a traditional initialisation file with name `filename` and writes a dictionary `obj` of key~value pairs to it. If values are not tables, they are written at the beginning of the file. If values are tables of key~value pairs, then they are written to the corresponding sections.

By default, the function writes the entries and sections in ascending order. You may change the order of the sections and the specific sections to be written by passing a table array of section names with the sections option, e.g. `sections=['Salad', 'Pizza']` first writes all entries of the Salad section, and then the Pizza section is written.

An optional spacer in front and behind the equals signs may be given by passing the spacer option which accepts any string, e.g. `spacer='\t'`. Default is the empty string.

A floating point value may be written with a decimal comma instead of a decimal dot by passing the `comma=true` option, default is `comma=false`.

The function returns nothing, is written in Agena and included in the lib/library.agn file.

See also: **utils.readini**.

utils.writexml (obj, filename [, indent])

Creates an XML file with name `filename` from the dictionary `obj` which should be of the same format as the dictionary returned by **utils.decodexml**.

The function returns nothing, is written in Agena and included in the lib/library.agn file.

See also: **utils.decodexml**, **utils.encodexml**, **utils.readxml**.

16.2 skycrane - Auxiliary Functions

As a *plus* package, the **skycrane** package is not part of the standard distribution and must be activated with the **import** statement, e.g. `import skycrane`.

The package contains functions that you might or might not find usefully.

`skycrane.bagtable (o)`

Creates a table of empty bags with its keys determined by the values in the sequence `o`. `o` may include values of any type. If `o` is empty, an error will be issued.

The function automatically loads the **bags** package if it has not yet been initialised.

The function is written in Agena and included in the lib/skycrane.agn file.

See also: **bags.bag**.

`skycrane.dice ()`

Returns random integers in the range [1 .. 6].

See also: **math.random**, **math.randomseed**.

`skycrane.enclose (str [, d])`

Encloses a string `str` with the given character or string `d`. If `d` is not given, the string is enclosed in double quotes. If `str` is a number, **null** or a Boolean, it is converted to a string before the operation starts. Otherwise it returns an error. It also returns an error if the optional second argument is not a string.

See also: **skycrane.removedquotes**.

`skycrane.fcopy (a, b [, verbose])`

This function is an interface to **os.fcopy** but can also deal with directories. If `a` and `b` are file names, then the function works like **os.fcopy**. If `b` is a directory, then `a` is copied into it. If `a` is a directory, then all files in it are copied into `b`.

If `verbose` is true then the name of the file copied successfully is printed at stdout.

The function is written in Agena and included in the lib/skycrane.agn file.

See also: **os.fcopy**, **skycrane.move**.

`skycrane.formatline (l [, ···])`

Similar to **io.writeline**, but all the strings, numbers or Booleans to be formatted must be passed in a table, sequence or register `l`. The function returns a string instead of writing the values to a file. The function accepts the following options:

- `delim =` <string>
  the delimiter, a semicolon by default, can also be the empty string;
- `enclose =` <string>
  each value will be enclosed by <string>, double quotes by default, can also be the empty string.

Example:

```
> skycrane.formatline([1, 'agena', true], delim = '|', enclose = '\''):
'1'|'agena'|'true'
```

`skycrane.getlocales ()`

Returns all locales available on your operating system. The return is a table with the keys being valid arguments to **os.setlocale**, and the entries the result of the respective call to **os.setlocale**.

The function is very, very slow, as **os.setlocale** is called around 460.000 times. In UNIX, it would be better to issue the command 'locale -a' in a shell to determine the locales supported on your system.

The function is written in Agena and included in the lib/skycrane.agn file.

See also: **os.setlocale**.

`skycrane.isemail (str [, strict])`

Checks whether a string `str` represents a valid E-mail address and returns **true** or **false**. The algorithm used does not cover all rules defined in RFC 3696, but should suffice with standard E-mail addresses of syntax "local-part@domain".

Note that the domain may not necessarily include a dot. You can override this rule by passing any non-**null** value for `strict`.

`skycrane.iterate (o)`

Returns an iterator function traversing a table, set, sequence or register `o` always in strict ascending order.

If `o` is a table, the function first sorts its keys and returns a function which if called, returns the table values of `o` in the ascending order of these sorted keys.

If `o` is a set, the function first sorts its entries and returns a function that if called, returns the elements one by one in ascending sorted order.

Although unnecessary: if `o` is a sequence or register, the function returns a function that if called, returns each value in `o` one by one in their original order.

The function is written in Agena and included in the lib/skycrane.agn file. For the order how keys or values will be sorted, see **sorted**.

A note: This function is utterly slow compared to the **for/in** statement. But there may be few situations demanding loops iterating in the strict ascending order of its (numeric or string) indices, or set, register, and sequence values.

See also: **ipairs**, **next**, **sorted**, **factory.iterate**, **factory.count**.

`skycrane.move (a, b [, verbose])`

This function is an interface to **os.move** but can also deal with directories. If `a` and `b` are file names, then the function works like **os.move**. If `b` is a directory, then `a` is moved into it. If `a` is a directory, then all files in it are moved into `b`.

The function is written in Agena and included in the lib/skycrane.agn file.

If `verbose` is true then the file copied successfully moved is printed at stdout.

See also: **os.move**, **skycrane.fcopy**.

`skycrane.readcsv (filename [, ···])`

Like **utils.readcsv**, but with the following default options, which can be overridden:

convert=false, ignorespaces=false, remove='doublequotes'.

The function is written in Agena and included in the lib/skycrane.agn file.

`skycrane.removedquotes (str)`

Removes enclosing double quotes from the string `str` and returns the modified string. If `str` is not enclosed by double quotes, `str` will be returned unmodified.

See also: **skycrane.enclose**.

`skycrane.replaceinfile (fn, oldstring, newstring [, true])`
`skycrane.replaceinfile (fn, subslist [, true])`

In the first form, replaces all occurrences of string `oldstring` in file `fn` (a filename) in-place with string `newstring`. The function supports pattern matching.

In the second form, one or more substitutions can be given by passing a table or sequence `sublist` of pairs of the form `oldstring` : `newstring`.

By default, a backup of the file to be modified is created with the additional suffix `` `.bup` ``. You can suppress any backup by passing the Boolean value **true** as the very last argument.

The function is written in Agena and included in the lib/skycrane.agn file.


```
skycrane.scribe (fh, obj [, ···])
skycrane.scribe (obj [, ···])
skycrane.scribe (···)
```

Like **io.write** and **io.writeline**, but if a table, sequence or register `obj` is being passed, it writes the values in the structure to the file denoted by its handle `fh` (first form) or the console (second form) instead of throwing an exception. `fh` is a file handle, not a file name.

The values in the structure `obj` must either be numbers or strings.

The function accepts the following options of type pair:

- If the `delim` option (third to last argument) has been passed, all values are separated by the given string. Default is a semicolon. Examples: `delim='|'`: use a pipe instead of a semicolon, `delim=''` (i.e. the empty string): do not include a delimiter.

- If the `newline` or `nl` option has been passed, and if its value is **false**, then no newline is included after the elements have been written. (Include a trailing delimiter - if needed - by calling **io.write**.) Default is **true**. Example: `newline=false`.

If no structure has been passed (third form), the function just behaves like **io.write** or **io.writeline**.

Examples:

```
> import skycrane;

> skycrane.scribe('men ne cunnon hwyder helrunan hwyrftum scriþað'):
men ne cunnon hwyder helrunan hwyrftum scriþað

> fd := io.open('Depeche Mode','wb');

> skycrane.scribe(fd,
>    'Enjoy the silence,
>     words are very unnecessary,
>     they can only do harm.');

> io.close(fd);
```

```
> fd := io.open('c:/wulfila.txt', 'w');

> paternoster³² := (/
>    atta unsar þu in himinam
>    weihnai namo þein
>    qimai þiudinassus þeins
>    wairþai wilja þeins
>    swe in himina jah ana airþai
>    hlaif unsarana þana sinteinan
>    gif uns himma daga \);

> skycrane.scribe(fd, paternoster, delim = ' ');

> io.close(fd);
```

The function is written in Agena and included in the lib/skycrane.agn file.

See also: **print**, **printf**, **io.write**, **io.writeline**, **skycrane.tee**.

skycrane.sorted (obj [, f])

Sorts a table, sequence or register `obj` non-destructively but contrary to **sort** and **sorted** can cope with structures including values of different types. First, numbers are sorted, then strings, the others are not. The function, however, is slower than **sorted**.

If `f` is given, then it must be a function that receives two structure elements, and returns **true** when the first is less than the second (so that not `f(obj[i+1], obj[i])` will be **true** after the sort). If `f` is not given, then the standard operator `<` (less than) is used instead.

The function is written in Agena and included in the lib/skycrane.agn file.

See also: **sort**, **sorted**, **stats.issorted**, **stats.sorted**.

skycrane.stopwatch ()

Implements a stopwatch. Just follow the instructions when calling skycrane.stopwatch(). The function returns nothing.

The function is written in Agena and included in the lib/skycrane.agn file.

See also: **watch**.

skycrane.tee (fh, x [,···] [, 'delim':str])

skycrane.tee (fh, x [,···], 'format':str)

In the first form, the function writes one or more numbers or strings `x` to both the console (stdout), and a file denoted by its handle `fh` to the current working directory. By default, the values are separated with a tabulator (\t). It finally puts a

---

³² Taken from the Gothic Language Wulfila Bible edited by Wilhelm Streitberg.

line feed at the end of the output. By passing the option `'delim':str`, as the last argument, the delimiter is given by the string `str`.

In the second form, one or more numbers or strings `x` are written to both the console (stdout), and a file denoted by its handle `fh` to the current working directory. The resulting string is formatted according to the printf-like template information in `str` passed with the `format` option. See **strings.format** for more information on the template string. It does not put a line feed at the end of the output, but to do so, you may add a `\n` control character to the end of the format string.

The function returns nothing.

The function is written in Agena and included in the lib/skycrane.agn file.

See also: **print**, **printf**, **skycrane.scribe**.

skycrane.timestamp (ts [, options])

Transforms the timestamp string `ts` of the format `'DD.MM.YYYY  HH:MM:SS'` or `'YYYY.MM.DD HH:MM:SS'` into a numeric Lotus Serial Date (LSD, a number) and - optionally - the time in DMS notation (a number, see **math.dms** for details).

`ts` does not need to include the number of minutes, hours and seconds. In this case, the missing digits are replaces by zeros. In `ts`, year and day may be swapped, and the month, day, hour, minute and second may be single integers, thus not necessarily preceded by a zero. Seconds can include fractional milliseconds.

If the option `splitup`=**true** is being passed, the function also returns the LSD as a string with a comma as the decimal separator, plus six integers depicting year, month, day, hour, minute and second. See third example below.

The function by default returns the time in Daylight Saving Time if active. You can switch this off (always returning Standard Time) by passing the option `standardtime`=**true**.

The delimiter in `ts` which separates year, month, day, a dot by default, can be changed to another delimiter by passing the `datedelim`=<any character> option, see example below.

The delimiter in `ts` which separates hour, minute and second, a colon by default, can be changed to another delimiter by passing the `timedelim`=<any character> option.

The delimiter in `ts` which separates date and time, a space by default, can be changed to another delimiter by passing the `datetimedelim`=<any character> option.

Optional delimiters may be preceded by a backslash so that the function can parse `ts` successfully.

```
> skycrane.timestamp('31.12.2017 23:59:01', datedelim='\.'):
43100.99931713  23.5901

> skycrane.timestamp('01/02/2017 23:59:01', datedelim='\/'):
42767.99931713  23.5901

> skycrane.timestamp('31-12-2017', datedelim='-'):
43100   0

> skycrane.timestamp('2017.2.1 23:59:01.999',
>    datedelim='\.', splitup=true):
42767.99931713  23.5901  42767,99931713  2017   2   1   23   59   1.999
```

The function is written in Agena and included in the lib/skycrane.agn file.

See also: **math.dms**, **os.lsd**, **skycrane.tocomma**, **skycrane.todate**.

skycrane.tocomma (x)

If x is a number, the function converts x to a string. If x is a float (containing a decimal dot), the dot is replaced by a comma. If x is a string and represents an integer or float, an optional decimal-dot is replaced by a comma.

The return is a string.

skycrane.todate (x)

Returns the calendar date and time represented by the number x, which should hold the number of seconds (and optionally milliseconds) elapsed since the start of the given epoch. The return is a string of the format `YYYY/MM/DD hh:mm:ss`.

If no argument is given, the current system date and time will be returned. You may pass an optional format string if you prefer another representation of the date and time.

See also: **strings.format**, **os.now**, **os.time**, **skycrane.timestamp**.

skycrane.tolerance (x, a)

Returns **math.branch**(**ceil**(a * **log10**(x + 1) - 1)), a maximum tolerance value especially suited for comparing similar strings where x may denote the size of a string. A good value for a might be a number greater than 3.

See also: **strings.dleven**, **strings.diffs**, **strings.fuzzy**.

`skycrane.trimpath (str)`

Converts backslashes in the string `str` to slashes and then removes, if existing, one trailing slash, and returns the modified string. If `str` does not include backslashes or trailing slashes/backslashes, `str` will be returned unmodified.

`skycrane.xmlmatch (str, tag [, tag₂, ···])`

Like **strings.match**, but returns the contents of the string `str` enclosed by the given XML search tag '<`tag`>(.-)</`tag`>'. If further tags tag_2 .. tag_k are given, then the result of the previous search with tag_{k-1} is checked for tag_k. If `tag` is a number, it will be converted to a string before matching. If any tag is not a string or number, it will be ignored.

The function returns **null** if any tag could not be found.

Examples:

```
> data := '<data>
>            <name>abc</name>
>            <info>
>                <name>def</name>
>            </info>
>        </data>'

> skycrane.xmlmatch(data, 'name'):
abc

> skycrane.xmlmatch(data, 'info', 'name'):
def
```

16.3 factory - Iterators

As a *plus* package, the **factory** package is not part of the standard distribution and must be activated with the **import** statement, i.e. `import factory`.

The package provides functional programming-style iterators.

`factory.count ([start [, step [, stop [, method]]]])`

Returns an iterator function that, each time it is called, returns a new number.

If no argument is given, the first number returned by the iterator is 0, the next call returns 1, the next one 2, and so forth. This means that the number returned with each call is increased by 1.

If only `start` is given, the first number returned by the iterator is `start`, the next call returns `start + 1`, the next one `start + 2`, and so forth. This means that the number returned with each call is increased by 1.

If `start` and `step` are given, the first number returned by the iterator is `start`, the next call returns `start + step`, the next one `start + 2*step`, and so forth. This means that the number returned with each call is increased by `step`, which may be negative. In the latter case the next number returned will be less than the current returned number.

If `stop` is given, the iterator returns **null** if the counter value exceeds `stop`. Default is +**infinity**.

If `start` or `step` are not numbers, the factory issues an error.

If `start` or `step` is a non-integer, the function by default automatically applies Kahan-Babuška summation to avoid round-off errors. You can choose between the following summation `method`s:

method	algorithm
`'babuska'`	Kahan-Babuška summation, highest accuracy but slowest (default)
`'kbn'`	Kahan-Babuška-Neumaier compensated summation, used in the Julia programming language
`'neumaier'`	Neumaier summation, good accuracy and performance
`'ozawa'`	Kahan-Ozawa summation

The generator automatically adds **hEps** to the `stop` value to avoid the iterator from leaving prematurely. If the absolute value of the `step` size is less than or equal to **hEps**, the generator will issue an error. You can entirely switch off this feature by setting **math.Eps** to zero, but only by calling **environ.kernel**:

```
> environ.kernel(hEps = 0);
```

The current setting of **hEps** can be queried by:

```
> environ.kernel('hEps'):
0
```

Note that **hEps** also controls numeric **for** loops with fractional step sizes. You might want to reset its value after generating the iterator to its default.

Example:

```
> import factory;

> f := factory.count(1, -0.1, -1, 'ozawa');  # count down

> while c := f() do print(c) od;
```

See also: **factory.reset**, **math.accu**, **skycrane.iterate**, **utils.uuid**.

factory.cycle (obj [, firstkey [, sentinel]])

Like **factory.iterate**, but when `sentinel` is encountered during traversal or the end of the structure has been reached, it does not return **null** but simply restarts iteration with the first element in `obj`.

See also: **next**, **factory.reset**.

factory.iterate (obj [, firstkey [, sentinel]])

Creates a function that when called, starting with index `firstkey`, iterates each element in a table, sequence or register `obj`, or each character in a string `obj` one by one, returning the respective index and value as two results.

`firstkey` and `sentinel` by default are **null**. With tables, `firstkey` may be any value and should be **null** if the table is to be iterated from its beginning. With any other data type, `firstkey` is a positive integer. If `sentinel` is encountered during traversal, the iterator returns **null**.

If there is nothing left, the function returns **null** and if called again re-starts iteration.

With strings, sequences and registers, if `firstkey` is out of range, the iterator simply returns **null**. With tables, if `firstkey` is a non-existent key, an error will be issued.

See also: **next**, **factory.cycle**, **factory.reset**, **skycrane.iterate**.

`factory.reset (f, index)`

Sets the current index of the iterator `f` created by **factory.cycle**, **factory.iterate** or **factory.count** to `index`.

If you want to re-iterate an object from its beginning, pass **null** for `index` if you traverse tables, and number 1 with every other object.

Chapter **Seventeen**

C API Functions

17 C API Functions

As already noted in Chapter 1, Agena features the same C API as Lua 5.1 so you are able to easily integrate your C packages and functions written for Lua 5.1 in Agena. Actually, Agena's C API is a superset of Lua's C API[33]. For a description of the API functions taken from Lua, see its Lua 5.1 manual. C API compatibility functions for Lua 5.2, 5.3 and 5.4 have also been added to facilitate porting C functions from these Lua editions to Agena.

The functions listed cannot be used in your Agena procedures - they have been created to access Agena's features from within C code. It generally supports GCC 3.4.6 and above.

If you would like to compile a Lua C package for Agena, usually only the names of following header files have to be changed:

Lua Header File	Corresponding Agena Header File
lua.h	agena.h
lauxlib.h	agnxlib.h
lualib.h	agenalib.h
luaconf.h	agnconf.h

The following Agena-specific header files exist:

Agena Header	Functionality
agncfg.h	This file will be created when executing `make config`. It determines the Endianness of your system, extends C long ints to eight bytes, and determines the date and time for the Agena build. It is advised to not change the contents of this header file.
agncmpt.h	Establishes cross-platform compatibility for certain mathematical C functions, a few 64-bit C types, and functions to work with files beyond the 2 GBytes size limit. Applicable primarily to Solaris, but also Linux, OS/2, Windows, and GCC.
agnhlps.h	Provides C helper functions and definitions, primarily for file access, further 64-bit types, quicksort, IEEE, Endian, mathematical operations & constants, cross-platform keyboard access, and fast and secure string concatenation and search-and-replace functions. Useful to compile Agena on SPARCs, PPCs, other RISC systems, and also on Little Endian architectures, since the **binio** package, **read,** and **save** work in Big Endian mode.
agnt64.h, agnt64_c.h, agnt64_l.h	Year 2038-fix headers for 32-bit systems.
cephes.h	Interface to Stephen L. Moshier's mathematical functions.
rlhmath.h	API to exponential integral functions written by RLH.

[33] Full compatibility to Lua's API has been established with Agena 1.6.0 in May 2012.

Agena Header	Functionality
interp.h	Interface to Professor Brian Bradie's various interpolation and spline functions.
sofa.h	Interface to the IAU Standards of Fundamental Astronomy (SOFA) Libraries.
moon.h, sunriset.h	Miscellaneous astronomical C functions
xbase.h	Interface to dBASE III file support of the Shapelib library.
charbuf.h	Small string character buffer library
luasys.h	Nodir Temirkhodjaev's Lua System (LuaSys v1.8)

Agena features a macro **agn_Complex** which is a shortcut for complex double.

The following API functions have been added (see files `lapi.c` and `agena.h`):

agn_absindex

```
LUA_API int agn_absindex (lua_State *L, int index, int gettop)
```

Returns the absolute positive stack index number for a given non-zero index `i` and the number of arguments `gettop` passed to a function.

agn_arrayborders

```
void agn_arraytoseq (lua_State *L, int idx, size_t a[])
```

Returns the lowest and highest indices in the array part of the table at `idx`, where the lowest index may start at 1 (not 0 as in C). `a` must be an array of 2 slots. If `a`[0], `a`[1] are 0, there is no array part in the table.

agn_arraypart

```
void agn_arraypart (lua_State *L, int idx)
```

Pushes a table with all the values in the array part of the table at index `idx` onto the top of the stack.

agn_arraytoseq

```
void agn_arraytoseq (lua_State *L, lua_Number *a, size_t n)
```

Converts a numeric array `a` with `n` elements to a sequence and pushes it on the top of the stack.

agn_asize

```
size_t agn_asize (lua_State *L, int idx);
```

Returns the number of items actually currently stored to the array part of the table at stack index `idx`, using a linear method. See also: **agn_size**.

agn_borders

```
void agn_borders (lua_State *L, int idx, size_t a[]);
```

Returns the smallest and largest assigned index - in this order - in the array and hash part of a table, in two-element array a. If zeros are returned, the array and hash parts of the table are empty.

agn_ccall

```
agn_Complex agn_ccall (lua_State *L, int nargs);
```
(Non-ANSI)

```
agn_Complex agn_ccall (lua_State *L, int nargs,
   lua_Number *real, lua_Number *imag);
```
(ANSI)

There are two different versions of this API function available. The first form supports Non-ANSI versions of Agena, e.g. Solaris, OS/2, etc. The second form can be used in the ANSI versions of Agena (compiled with the LUA_ANSI option).

Non-ANSI version: Exactly like **lua_call**, but returns a complex value as its result, so a subsequent conversion to a complex number via stack operation is avoided. If the result of the function call is not a complex value, an error will be issued. **agn_ccall** pops the function and its arguments from the stack.

ANSI version: Like **lua_call**, but returns the real and imaginary parts of the complex result through the parameters `real` and `imag`. If the result of the function call is not a complex value, an error will be issued. **agn_ccall** pops the function and its arguments from the stack.

The function always returns the first result of the function call.

The function does *not* reserve its own stack space so you must call **lua_checkstack** or **luaL_checkstack** before.

agn_checkboolean

```
int agn_checkboolean(lua_State *L, int idx);
```

Checks whether the value at index `idx` is a Boolean and returns 1 for **true**, and 0 for **false** or **fail**. An error will be raised if the value at `idx` is none of them.

agn_checkcomplex

```
LUALIB_API agn_Complex agn_checkcomplex (lua_State *L, int idx)
```

Checks whether the value at index `idx` is a complex value and returns it. An error is raised if the value at `idx` is not of type complex.

agn_checkinteger

```
lua_Integer agn_checkinteger (lua_State *L, int idx);
```

Checks whether the value at index `idx` is a number and an integer and returns this integer. An error is raised if the value at `idx` is not a number, or if it is a float.

See also: agn_checknonposint, agn_checkposint.

agn_checklstring

```
const char *agn_checklstring (lua_State *L, int idx, size_t *len);
```

Works exactly like luaL_checklstring but does not perform a conversion of numbers to strings.

agn_checknonnegint

```
lua_Integer agn_checknonnegint (lua_State *L, int idx);
```

Checks whether the value at index `idx` is a number and a non-negative integer and returns this integer. An error is raised if the value at `idx` is not a number, or if it is a float or is negative.

See also: **agn_checkinteger**, **agn_checkposint**, **agn_checkuint16_t**, **agn_checkuint32_t**.

agn_checknonnegative

```
lua_Number agn_checknonnegative (lua_State *L, int idx);
```

Like **agn_checknumber**, but checks whether the number at `idx` is non-negative.

See also: **agn_checkpositive**.

agn_checknumber

```
lua_Number agn_checknumber (lua_State *L, int idx);
```

Checks whether the value at index `idx` is a number and returns this number. An error is raised if the value at `idx` is not a number. This procedure is an alternative to **luaL_checknumber** for it is around 14 % faster in execution while providing the same functionality by avoiding different calls to internal Auxiliary Library functions.

See also: **agn_checkpositive**, **agn_checknonnegative**.

agn_checkposint

```
lua_Integer agn_checkposint (lua_State *L, int idx);
```

Checks whether the value at index `idx` is a number and a positive integer and returns this integer. An error is raised if the value at `idx` is not a number, or if it is a float or is non-positive.

See also: **agn_checkinteger**, **agn_checknonnegint**.

agn_checkpositive

```
lua_Number agn_checkpositive (lua_State *L, int idx);
```

Like **agn_checknumber**, but checks whether the number at `idx` is positive.
See also: **agn_checknonnegative**.

agn_checkuint16_t

```
uint16_t agn_checkuint16_t (lua_State *L, int idx);
```

Checks whether its argument at stack position `idx` is an unsigned integer and whether it fits into the range $0 .. 2^{16} - 1$.

See also: **agn_checknonnegint**, **agn_checkuint32_t**, **agnL_optuint32_t**.

agn_checkuint32_t

```
uint32_t agn_checkuint32_t (lua_State *L, int idx);
```

Checks whether its argument at stack position `idx` is an unsigned integer and whether it fits into the range $0 .. 2^{32} - 1$.

See also: **agn_checknonnegint**, **agn_checkuint16_t**.

agn_checkstring

```
const char *agn_checkstring (lua_State *L, int idx);
```

Works exactly like luaL_checkstring but does not perform a conversion of numbers to strings. An error is raised if `idx` is not a string.

If `idx` is negative: due to garbage collection, there is no guarantee that the pointer returned will be valid after the corresponding value is removed from the stack.

agn_cleanse

```
LUA_API void agn_cleanse (lua_State *L, int idx, int gc)
```

Empties the entire contents of the table at index `idx`, but does not delete it. If `gc` is 1, then a garbage collection is performed, as well. Set it to 0 of no garbage collection shall be triggered. The function does not change the stack.

agn_cleanseset

```
LUA_API void agn_cleanseset (lua_State *L, int idx, int gc)
```

Empties the entire contents of the set at index `idx`, but does not delete it. If `gc` is 1, then a garbage collection is performed, as well. Set it to 0 of no garbage collection shall be triggered. The function does not change the stack.

agn_complexgetimag

```
LUA_API void agn_complexgetimag (lua_State *L, int idx)
```

Pushes the imaginary part of the complex value at position `idx` onto the stack.

agn_complexgetreal

```
LUA_API void agn_complexgetreal (lua_State *L, int idx)
```

Pushes the real part of the complex value at position `idx` onto the stack.

agn_compleximag

```
lua_Number agn_compleximag (lua_State *L, int idx)
```

Returns the imaginary part of the complex value at stack index `idx` as a lua_Number. See also: **agn_complexreal**.

agn_complexreal

```
lua_Number agn_complexreal (lua_State *L, int idx)
```

Returns the real part of the complex value at stack index `idx` as a lua_Number. See also: **agn_compleximag**.

agn_copy

```
LUA_API void agn_copy (lua_State *L, int idx, int mode)
```

Returns a true copy of the table, set, or sequence at stack index `idx`. The copy is put on top of the stack, but the original structure is not removed. `mode` controls what to do with tables: `mode` = 1: copy array part only; `mode` = 2: copy hash part only; `mode` = 3: copy both array and hash part. With all structures, if `mode` = 7 then no metatables or user-defined types are copied. The function performs a garbage collection.

agn_createcomplex

```
LUA_API void agn_createcomplex (lua_State *L, agn_Complex c)
```

Pushes a value of type complex onto the stack with its complex value given by `c`.

agn_createpair

```
void agn_createpair (lua_State *L, int idxleft, int idxright);
```

Pushes a pair onto the stack with the left operand determined by the value at index `idxleft`, and the right operand by the value at index `idxright`. The left and right values are *not* popped from the stack. The function performs a garbage collection.

agn_createpairnumber

```
void agn_createpairnumbers (lua_State *L, lua_Number l, lua_Number r);
```

Pushes a pair onto the stack with the left operand set to number `l`, and the right operand set to number `r`. The function performs a garbage collection.

agn_createreg

```
LUA_API void agn_createreg (lua_State *L, int nrec)
```

Pushes a register onto the top of the stack with `nrec` pre-allocated places (`nrec` may be zero).

agn_creatertable

```
LUA_API void agn_creatertable (lua_State *L, int idx)
```

Creates an empty remember table for the function at stack index `idx`. It does not change the stack.

agn_createseq

```
void agn_createseq (lua_State *L, int nrec);
```

Pushes a sequence onto the top of the stack with `nrec` pre-allocated places (`nrec` may be zero).

agn_createset

```
void agn_createset (lua_State *L, int nrec);
```

Pushes an empty set onto the top of the stack. The new set has space pre-allocated for `nrec` items.

agn_createtable

```
LUA_API void agn_createtable (lua_State *L, int narray, int nrec)
```

Like **lua_createtable**, but marks the new table such that the **size** operator will always return the correct number of elements stored in its array part. Note that **size** is slower on these special tables (arrays) since it has to conduct a linear count - instead of a binary one - on its array part.

agn_deletefield

```
LUA_API void agn_deletefield (lua_State *L, int idx, const char *key)
```

Deletes the field `key` from the table at index `idx` without invoking metamethods. The function leaves the stack unchanged.

agn_deletertable

```
LUA_API void agn_deletertable (lua_State *L, int objindex)
```

Deletes the remember table of the procedure at stack index `idx`. If the procedure has no remember table, nothing happens. The function leaves the stack unchanged.

agn_equalref

```
int lua_equalref (lua_State *L, int idx1, int idx2);
```

Compares any two values at stack indices `idx1` and `idx2`, and returns 0 if they are different and 1 if they are equal. See **environ.isequal** for more information. The function does not change the stack.

agn_entries

```
void agn_entries (lua_State *L, int idx, int *flag);
```

Returns all the values stored to the table at stack index `idx` in a new table and sets it to the top of the stack. `flag` is set to 0 if no value are residing in the hash part, and to 1 if there is at least one element in the hash part.

agn_fnext

```
int agn_fnext (lua_State *L, int indextable, indexfunction, int mode);
```

Pops a key from the stack, and pushes three or four values in the following order: the key of a table given by `indextable`, its corresponding value (if mode = 1), the function at stack number `indexfunction`, and the value from the table at the given `indextable`. If there are no more elements in the table, then **agn_fnext** returns 0 (and pushes nothing).

The function is useful to avoid duplicating values on the stack for **lua_call** and the iterator to work correctly.

A typical traversal looks like this:

```
/* table is in the stack at index 't', function is at stack index 'f' */
lua_pushnil(L); /* first key */
while (lua_fnext(L, t, f, 0) != 0) {
  /* 'key' is at index -3, function at -2, and 'value' at -1 */
  lua_call(L, 1, 1);  /* call the function with one arg & one result */
  lua_pop(L, 1);      /* removes result of lua_call;
                         keeps 'key' for next iteration */
}
```

While traversing a table, do not call **lua_tolstring** directly on a key, unless you know that the key is actually a string. Recall that **lua_tolstring** changes the value at the given index; this confuses the next call to **lua_next**.

agn_getbitwise

```
int agn_getbitwise (lua_State *L)
```

Returns the current mode for bitwise arithmetic: 0 if the bitwise operators (**&&**, **||**, **^^**, **~~**, and **shift**), internally calculate with unsigned integers, and 1 if signed integers are used.

See also: **agn_setbitwise**.

agn_getcmplxparts

```
void agn_getcmplxparts (lua_State *L, int idx,
   lua_Number *re, lua_Number *im)
```

Expects a number or complex number at stack position `idx` and returns its real and imaginary part in `re` and `im`. If the value at `idx` is a number or complex number, returns 1 and 0 otherwise. With numbers, `im` will always be 0.

agn_getconstants

```
int agn_getconstants (lua_State *L)
```

Returns a non-zero if constant declarations are active, and 0 otherwise.

See also: **agn_setconstants**.

agn_getdblepsilon

```
lua_Number agn_getdblepsilon (lua_State *L)
```

The macro returns the setting of the double-accuracy threshold epsilon, i.e. system variable "DoubleEps". See also: **agn_getepsilon**, **agn_setdblepsilon**.

agn_getduplicates

```
int agn_getduplicates (lua_State *L)
```

Returns a non-zero if parser duplicate declaration warnings are turned on, and 0 otherwise.

See also: **agn_setduplicates**.

agn_getemptyline

```
int agn_getemptyline (lua_State *L)
```

Returns the current setting for two input prompts always being separated by an empty line and pushes a Boolean on the stack.

See also: **agn_setemptyline**.

agn_geteps

```
lua_Number agn_geteps (lua_State *L, const char *varname)
```

Returns the value of the Agena system variable `Eps` (epsilon) if varname is "Eps", and that of DoubleEps if varname is "DoubleEps", without changing the stack.

agn_getepsilon

```
lua_Number agn_getepsilon (lua_State *L)
```

The macro returns the setting of the accuracy threshold epsilon used by the ~= operator and the **approx** function. See also: **agn_getdblepsilon**, **agn_setepsilon**.

agn_getfunctiontype

```
LUA_API int agn_getfunctiontype (lua_State *L, int idx)
```

Returns 1 if the function at stack index idx is a C function, 0 if the function at idx is an Agena function, and -1 of the value at idx is no function at all.

agn_gethepsilon

```
lua_Number agn_gethepsilon (lua_State *L)
```

The macro returns the setting **hEps**. See also: Chapter 5.2.2 and **agn_sethepsilon**.

agn_getinumber

```
lua_Number agn_getinumber (lua_State *L, int idx, int n);
```

Returns the value t[n] *as a* lua_Number, where t is a table at the given valid index `idx`. If t[n] is not a number, the return is 0. The access is raw; that is, it does not invoke metamethods.

agn_getistring

```
const char *agn_getistring (lua_State *L, int idx, int n);
```

Returns the value t[n] as *a const char* *, where t is a table at the given valid index idx. If t[n] is not a string, the return is NULL. The access is raw; that is, it does not invoke metamethods.

agn_getlibnamereset

```
int agn_getlibnamereset (lua_State *L)
```

Returns the current setting for the **restart** statement to also reset **libname** and either pushes a non-zero integer (= **true**) or zero (= **false**).

See also: **agn_setlibnamereset**.

agn_getlongtable

```
int agn_getlongtable (lua_State *L)
```

Returns the current setting for key~value pairs in tables being output line by line instead of just a single line and puts a Boolean on the stack. A non-zero integer denotes the feature is switched on, and 0 it is switched off.

See also: **agn_setlongtable**.

agn_getround

```
LUA_API void agn_getround (lua_State *L)
```

Gets the current rounding mode. Pushes the string "downward" for FE_DOWNWARD, "upward" for FE_UPWARD, "nearest" for FE_TONEAREST, and "towardzero" for FE_TOWARDZERO onto the stack. If the rounding mode could not be determined, **undefined** is pushed. If any other FE_* value is determined, **fail** will be pushed. Not available in DOS.

See also: **agn_setround**.

agn_getrtable

```
LUA_API int agn_getrtable (lua_State *L, int idx)
```

Pushes the remember table of the function at stack index `idx` onto the stack and returns 1. If the function does not have a remember table, it pushes nothing and returns 0. See also: **agn_getstore**.

agn_getrtablewritemode

```
int agn_getrtablewritemode (lua_State *L, int idx)
```

Returns 0 if the remember table of the function at stack index idx cannot be updated by the **return** statement (i.e. if it is an rotable), 1 if it can (i.e. if it is an rtable), 2 if the function at idx has no remember table at all, and -1 if the value at idx is not a function.

agn_getseqlstring

```
const char *agn_getseqlstring (lua_State *L, int idx, int n, size_t *l);
```

Gets the string at index `n` in the sequence at stack index `idx`. The length of the string is stored to `l`.

agn_getstore

```
LUA_API int agn_getstore (lua_State *L, int idx)
```

Pushes the internal 'store' table of the function at stack index `idx` onto the stack and returns 1. If the function does not have a store, it pushes nothing and returns 0. See also: Chapter 6.25 and **agn_getrtable**.

agn_getutype

```
int agn_getutype (lua_State *L, int idx);
```

Returns the user-defined type of a procedure, table, sequence, set, userdata, or pair at stack position `idx` as a string, pushes it onto the top of the stack and returns 1. If no user-defined type has been defined, the function returns 0 and pushes nothing onto the stack.

See also: **agn_isutype**, **agn_setutype**.

agn_hashpart

```
void agn_hashpart (lua_State *L, int idx)
```

Pushes a table with all the values in the hash part of the table at index `idx` onto the top of the stack.

agn_integerindices

```
void agn_integerindices (lua_State *L, int idx, int *flag);
```

Returns all integer indices of the table t at stack index `idx` in a new table and puts it onto the top of the stack. It also sets `flag` to 0 if there are no integer indices in hash part of t, and `flag` to 1 if there is at least one integer index in the hash part.

agn_isfail / lua_isfail

```
int agn_isfail (lua_State *L, int idx);
int lua_isfail (lua_State *L, int idx);
```

Returns 1 if the Boolean value at the given acceptable index results to **fail**, 0 otherwise (**true** and **false**). **lua_fail** first checks whether the value at the index is a

Boolean to avoid crashes if it is not, and then whether it represents **fail**. The functions actually are C macros. See also: **agn_istrue**, **agn_isfail**, **lua_istrue**, **lua_isfalse**.

agn_isfalse / lua_isfalse

```
int agn_isfalse (lua_State *L, int idx);
int lua_isfalse (lua_State *L, int idx);
```

Returns 1 if the Boolean value at the given acceptable index results to **false**, 0 otherwise (**true** and **fail**). **lua_false** first checks whether the value at the index is a Boolean to avoid crashes if it is not, and then whether it represents **false**. The function actually are C macros. See also: **agn_istrue**, **agn_isfail**, **lua_istrue**, **lua_isfail**.

agn_isfloat

```
agn_isfloat (lua_State *L, int idx);
```

Returns 1 if the value at the given acceptable index is a number but not an integral, and 0 otherwise. See also: **agn_isinteger**, **agn_isnumber**.

agn_isinteger

```
agn_isinteger (lua_State *L, int idx);
```

The function returns 1 if the value at the given acceptable index is a number representing an integer, and 0 otherwise. See also: **agn_isnumber**.

agn_islinalgvector

```
int agn_islinalgvector (lua_State *L, int idx, size_t *dim)
```

Tests if a value at the given acceptable index is a vector created with the **linalg** package, and returns 1 if true and 0 otherwise. It also stores the dimension of the vector in `dim`.

agn_isnumber

```
agn_isnumber (lua_State *L, int idx);
```

This macro returns 1 if the value at the given acceptable index is a number, and 0 otherwise. See also: **agn_isfloat**, **agn_isinteger**.

agn_issequtype

```
int *agn_issequtype (lua_State *L, int idx, const char *str);
```

Checks whether the type at stack index `idx` is a sequence and whether the sequence has the user-defined type denoted by `str`. It returns 1 if the above condition is true, and 0 otherwise.

agn_issetutype

```
int *agn_issetutype (lua_State *L, int idx, const char *str);
```

Checks whether the type at stack index `idx` is a set and whether this set has the user-defined type denoted by `str`. It returns 1 if the above condition is true, and 0 otherwise.

agn_isstring

```
agn_isstring (lua_State *L, int idx);
```

This macro returns 1 if the value at the given acceptable index `idx` is a string, and 0 otherwise.

agn_istableutype

```
int *agn_istableutype (lua_State *L, int idx, const char *str);
```

Checks whether the type at stack index `idx` is a table and whether the table has the user-defined type denoted by `str`. It returns 1 if the above condition is true, and 0 otherwise.

agn_istrue / lua_istrue

```
int agn_istrue (lua_State *L, int idx);
int lua_istrue (lua_State *L, int idx);
```

Returns 1 if the Boolean value at the given acceptable index `idx` results to **true**, 0 otherwise (**false** and **fail**). **lua_true** first checks whether the value at the index is a Boolean to avoid crashes if it is not, and then whether it represents **true**. The functions actually are C macros. See also: **agn_isfalse**, **agn_isfail**, **lua_isfalse**, **lua_isfail**.

agn_isutype

```
int *agn_isutype (lua_State *L, int idx, const char *str);
```

Checks whether a user-defined type `str` has been set for the given table, set, sequence, pair, or procedure at stack position `idx`. It returns 1 if the user-defined type has been set, and 0 otherwise.

agn_isutypeset

```
int *agn_isutypeset (lua_State *L, int idx, const char *str);
```

Checks whether a user-defined type has been set for the given object at stack position `idx`. It returns 1 if a user-defined type has been set, and 0 otherwise. The function accepts any Agena types. By default, if the object is not a table, sequence, a pair, set, or procedure, it returns 0.

agn_malloc

```
void *agn_malloc (lua_State *L, size_t size, const char *procname, ...);
```

Allocates `size` bytes of memory and returns a pointer to the newly allocated block. In case memory could not be allocated, it returns an error message including `procname` that called **agn_malloc**. The function optionally can free one or more objects referenced by their pointers in case memory allocation failed.

In all cases, the last argument must be **NULL**.

See also: **agn_free**.

agn_ncall

```
lua_Number agn_ncall (lua_State *L, int nargs, *int error, int quit);
```

Exactly like **lua_call**, but returns a numeric result as an Agena number, so a subsequent conversion to a number via stack operations is avoided. If the result of the function call is not numeric, an error will be issued. **agn_ncall** pops the function, its arguments and the result from the stack, leaving it leveled. It always returns the first result of the function call.

If the function call does not evaluate to a number, `error` is set to 1 and 0 otherwise. If `quit` is 1, the function will automatically issue an error if the result is not a number; otherwise `quit` should be set to 0. The function does *not* allocate its own stack space, so you must call **lua_checkstack** or **luaL_checkstack** before.

agn_nops

```
size_t agn_nops (lua_State *L, int idx);
```

Determines the number of actual table, set, sequence or register entries of the structure or the size of a string at stack index `idx`. If the value at `idx` is not one of the mentioned data types, it returns 0. With tables, this procedure is an alternative to **lua_objlen** if you want to get the size of a table since **lua_objlen** does not return correct results if there are holes in the table or if the table is a dictionary.

agn_onexit

```
LUA_API void agn_onexit (lua_State *L)
```

Pushes the function **environ.onexit** if it exists, and calls it. The function leaves the stack unchanged.

agn_optcomplex

```
agn_Complex agn_optcomplex (lua_State *L, int narg, agn_Complex z);
```

If the value at index `narg` is a complex number, it returns this number. If this argument is absent or is **null**, the function returns complex `z`. Otherwise, raises an error.

agn_pairgeti

```
void agn_pairgeti (lua_State *L, int idx, int n);
```

Returns the left operand of a pair at stack index `idx` if `n` is 1, and the right operand if `n` is 2, and puts it onto the top of the stack. You have to make sure that `n` is either 1 or 2.

agn_pairrawget

```
void agn_pairrawget (lua_State *L, int idx);
```

Pushes onto the stack the left or the right hand value of a pair `t`, where `t` is the value at the given valid index `idx` and the number `k` (`k=1` for the left hand side, `k=2` for the right hand side) is the value at the top of the stack. It does not invoke any metamethods. This function pops both `k` from the stack.

agn_pairrawset

```
void agn_pairrawset (lua_State *L, int idx);
```

Does the equivalent to p[k] := v, where p is a pair at the given valid index `idx`, v is the value at the top of the stack, and k is the value just below the top.

This function pops both the key and the value from the stack. It does not invoke any metamethods. See also: **agn_pairset**, **agn_pairseti**.

agn_pairset

```
LUA_API void agn_pairset (lua_State *L, int idx, int idxleft, int idxright)
```

Sets the two values at indices `idxleft` and `idxright` to the pair at index `idx`. The function does not pop the values.

See also: **agn_pairrawset**, **agn_pairseti**.

agn_pairseti

```
LUA_API void agn_pairseti (lua_State *L, int idx, int pos)
```

Sets the value at the stack top to the pair at index `idx` and pops the value. If `pos` is 1, then the value is put into the left part of the pair, if `pos` is 2, then the right part is set.

See also: **agn_pairrawset**, **agn_pairset**.

agn_pairstate

```
LUA_API void agn_pairstate (lua_State *L, int idx, size_t a[])
```

Returns a flag indicating whether a metatable has been assigned to the pair at index `idx` in a, a C array with one entry, where 1 indicates that the pair features a metatable, and 0 means it does not.

agn_parts

```
void agn_parts (lua_State *L, int idx)
```

Pushes two tables onto the top of the stack: one with all the values in the array part of the table at index `idx`, and one with the values in its hash part.

agn_poptop

```
void agn_poptop (lua_State *L);
```

Pops the top element from the stack. The function is more efficient than lua_pop(L, 1).

agn_poptoptwo

```
void agn_poptoptwo (lua_State *L);
```

Pops the top element and the value just below the top from the stack. The function is more efficient than lua_pop(L, 2).

agn_pushboolean

```
void agn_pushboolean (lua_State *L, int b);
```

Pushes true onto the stack if `b` is 1 or larger, and pushes false onto the stack if `b` is 0. If `b` is -1, it pushes fail onto the stack.

agn_pushcomplex

```
void agn_pushcomplex (lua_State *L, lua_Number re, lua_Number im);
```

Pushes the complex value `re` + I*`im` onto the stack. The macro can be used both with Agena versions using standard complex.h complex functions and those using proprietary complex arithmetic.

agn_rawgetfield

```
LUA_API void agn_rawgetfield (lua_State *L, int idx, const char *field)
```

Returns t[`field`], where table t resides at stack index `idx`.

agn_rawgetifield

```
LUA_API void agn_rawgetifield (lua_State *L, int idx, int kidx)
```

Returns t[i], where table t resides at stack index `idx` and the key i at stack index `kidx`.

agn_rawgetinumber

```
lua_Number agn_rawgetinumber (lua_State *L, int idx, int n, int *rc);
```

Returns the value t[n] *as a* lua_Number, where t is a table at the given valid index `idx`. If t[n] is not a number, the return is 0. The access is raw; that is, it does not invoke metamethods. `rc` includes the result of the retrieval and is 1 on success and 0 otherwise, i.e. t[n] is not a number or a string convertible to a number.

agn_rawinsert

```
LUA_API void agn_rawinsert (lua_State *L, int idx)
```

Inserts the value at the top of the stack to the table at index `idx`, more precisely, it is added to the end of the array part of the table. The value is popped from the stack.

agn_rawinsertfrom

```
LUA_API void agn_rawinsertfrom (lua_State *L, int tidx, int vidx)
```

Inserts the value at stack index `vidx` to the table residing at index `idx`, more precisely, it is added to the end of the array part of the table.

If `vidx` = -1, then the value is popped from the stack, otherwise the stack is left untouched.

agn_rawreggetinumber

```
lua_Number agn_rawreggetinumber (lua_State *L, int idx, int n, int *rc);
```

Returns the value t[n] *as a* lua_Number, where t is a register at the given valid index `idx`. If t[n] is not a number, the return is 0. The access is raw; that is, it does not invoke metamethods. `rc` includes the result of the retrieval and is 1 on success and 0 otherwise, i.e. t[n] is not a number or a string convertible to a number.

agn_rawseqgetinumber

```
lua_Number agn_rawseqgetinumber (lua_State *L, int idx, int n, int *rc);
```

Returns the value t[n] *as a* lua_Number, where t is a sequence at the given valid index `idx`. If t[n] is not a number, the return is 0. The access is raw; that is, it does not invoke metamethods. `rc` includes the result of the retrieval and is 1 on success and 0 otherwise, i.e. t[n] is not a number or a string convertible to a number.

agn_rawsetfield

```
LUA_API void agn_rawsetfield (lua_State *L, int idx, const char *field)
```

Does the equivalent to t[`field`] := v, where t is a table at the given valid index `idx`, and v is the value at the top of the stack, without invoking metamethods.

This function pops the value from the stack. It does not invoke any metamethods.

See also: **agn_deletefield**.

agn_regextend

```
LUA_API int agn_regextend (lua_State *L, int idx, size_t newsize)
```

Extends the size of the register at stack position `idx` to `newsize` elements and fills the newly created slots with **null**. If `newsize` is less than the current size, it simply returns 0 and does not change the size of the register, otherwise the function returns 1. If the current top pointer already refers to the total size of the register, it is set to `newsize`, otherwise it is left unchanged.

agn_reggeti

```
LUA_API void agn_reggeti (lua_State *L, int idx, size_t n)
```

Pushes the value stored at position `n` of the register located at stack index `idx` to the top of the stack. If `n` is out-of-range, or larger than the position of the top pointer, it issues an error.

agn_reggetinumber

```
LUA_API lua_Number agn_reggeti (lua_State *L, int idx, size_t n)
```

The function the number stored at position `n` of the register located at stack index `idx`. If `n` is out-of-range, or larger than the position of the top pointer, it issues an error. It returns **infinity** if the value at `n` is non-numeric.

agn_reggettop

```
LUA_API size_t agn_reggettop (lua_State *L, int idx)
```

Returns the position of the top pointer of a register at stack index `idx`. See also: **agn_regsettop**.

agn_regpurge

```
LUA_API void agn_regpurge (lua_State *L, int idx, int n)
```

Removes the value at position `n` of the register at stack index `idx` and shifts down all values beyond `n` if necessary. The function does not reduce the size of the register, but decrements the top pointer by 1.

agn_regrawget

```
LUA_API void agn_regrawget (lua_State *L, int idx)
```

Pushes onto the stack the value t[k], where t is the register at the given valid index `idx` and `k` is the value at the top of the stack.

This function pops the key from the stack (putting the resulting value in its place). It does not invoke metamethods.

agn_regreduce

```
LUA_API int agn_regreduce (lua_State *L, int idx, size_t newsize, int nil)
```

Reduces the size of the register residing at stack index `idx` to `newsize` entries. If `nil` is 1, then all values residing at positions larger then `newindex`, are **null**'ed, otherwise set `nil` to 0. The function returns 0 if `newindex` is less than 0, and 1 otherwise. See also: **agn_regextend**.

agn_regset

```
LUA_API void agn_regset (lua_State *L, int idx)
```

Assumes that the value to be set to a register residing at stack position `idx` is at the top of the stack and the numeric key just below the stack and conducts the assignment.

agn_regseti

```
LUA_API void agn_regseti (lua_State *L, int idx, int n)
```

Sets the value residing at the top of the stack to position `n` of the register at index `idx` and pops the inserted value from the stack.

agn_regsettop

```
LUA_API int agn_regsettop (lua_State *L, int idx)
```

Sets the current top pointer of a register residing at index `idx` to the number stored at the top of the stack. The number at the top of the stack is popped therafter. See also: **agn_reggettop**.

agn_regstate

```
LUA_API void agn_regstate (lua_State *L, int idx, size_t a[])
```

Returns the current top pointer, the total number of items, and a flag indicating whether a metatable has been assigned to the register at index `idx` in a, a C array with three entries. The position of the top pointer is stored to a[0], the total number of entries to a[1]. The metatable flag is stored to a[2], where 1 indicates that the sequence features a metatable, and 0 means it does not.

agn_seqrawgetinumber

```
lua_Number agn_seqrawgetinumber (lua_State *L, int idx, int n);
```

Returns the value t[n] *as a* lua_Number, where t is a sequence at the given valid index `idx`. If t[n] is not a number, the return is 0. The access is raw; that is, it does not invoke metamethods.

See also: **lua_seqrawgetinumber**.

agn_seqresize

```
int agn_seqresize (lua_State *L, int idx, size_t newsize);
```

Shrinks or expands the sequence at stack index `idx`, i.e. adds or deletes the number of pre-allocated slots to exactly `newsize`. The function takes care to nullify all surplus values before shrinking the sequence or adding **null**s when expanding it.

agn_seqsize

```
size_t agn_seqsize (lua_State *L, int idx);
```

Returns the number of items currently stored to the sequence at stack index `idx`.

agn_seqstate

```
void agn_seqstate (lua_State *L, int idx, size_t a[])
```

Returns the actual number of items, the maximum number of items assignable to, and a flag indicating whether a metatable has been assigned to the sequence at index `idx` in a, a C array with three entries. The actual number of items is stored to a[0], the maximum number of entries to a[1]. If a[1] is 0, then the number of possible entries is infinite. The metatable flag is stored in a[2], where 1 indicates that the sequence features a metatable, and 0 means it does not.

agn_setbitwise

```
void agn_setbitwise (lua_State *L, int value)
```

Sets the mode for bitwise arithmetic. If `value` is greater than 0, the bitwise functions (**&&**, **||**, **^ ^**, **~~**, and **shift**) internally calculate with signed integers, otherwise Agena calculates with unsigned integers.

See also: **agn_getbitwise**.

agn_setconstants

```
void agn_setconstants (lua_State *L, int value)
```

Switches on constants mode if `value` is non-zero, and switches it off if 0.

See also **agn_getconstants**.

agn_setdblepsilon

```
lua_Number agn_setdblepsilon (lua_State *L, lua_Number x)
```

Sets the double-accuracy threshold epsilon to system variable "DoubleEps". See also: **agn_setepsilon**.

agn_setduplicates

```
void agn_setduplicates (lua_State *L, int value)
```

Switches on duplicate declaration warnings (shadowing) if `value` is non-zero, and switches it off if 0.

See also **agn_getduplicates**.

agn_setemptyline

```
void agn_setemptyline (lua_State *L, int value)
```

If `value` is greater than 0, then two input prompts are always separated by an empty line. If set **false**, no empty line is inserted.

See also: **agn_getemptyline**.

agn_setepsilon

```
lua_Number agn_setepsilon (lua_State *L, lua_Number x)
```

Sets the accuracy threshold epsilon used by the ~= operator and the **approx** function to the number `x`. See also: **agn_getepsilon**.

agn_sethepsilon

```
lua_Number agn_sethepsilon (lua_State *L, lua_Number x)
```

Sets the **hEps** constant. See also: Chapter 5.2.2 and **agn_gethepsilon**.

agn_setlibnamereset

```
void agn_setlibnamereset (lua_State *L, int value)
```

If `value` is greater than 0, then the **restart** statement resets libname to its default. If `value` is non-positive, then libname is not changed with a **restart**.

See also: **agn_getlibnamereset**.

agn_setlongtable

```
void agn_setlongtable (lua_State *L, int value)
```

If `value` is greater than 0, then the **print** function outputs key~value pairs in tables line-by-line. If `value` is non-positive, then the print function prints all pairs in a single consecutive line.

See also: **agn_getlongtable**.

agn_setreadlibbed

```
int agn_setreadlibbed (lua_State *L, const char *name)
```

Inserts name into the global set **package.readlibbed**.

agn_setresize

```
void agn_setresize (lua_State *L, int idx, size_t newsize, int protect)
```

Resizes the set at stack index `idx` to `newsize` pre-allocated slots. `protect` controls whether to allow only a set to be shrunk without dropping any elements (`protect == 1`), shrunk or enlarged without dropping any elements (`protect == 2`), or

whether to have full control what may happen: shrinking or expanding, dropping or not dropping any elements (`protect` == 0). With `protect` == 0, it is advised that the set is empty.

agn_setround

```
int agn_setreadlibbed (lua_State *L, const char *name)
```

Sets the rounding mode. `what` may be "downward" for FE_DOWNWARD, "upward" for FE_UPWARD, "nearest" for FE_TONEAREST, and "towardzero" for FE_TOWARDZERO. Returns 1 on success, and 0 otherwise. In case of failure, the former rounding mode is re-established. Not available in DOS.

See also: **agn_getround**.

agn_setrtable

```
LUA_API void agn_setrtable (lua_State *L, int find, int kind, int vind)
```

Sets argument~return values to the function at stack index `find`. The argument list reside at a table array at stack index `kind`, the return list are in another table at stack index `vind`. See the description for the **rset** function for more information.

agn_setudmetatable

```
LUA_API void agn_setudmetatable (lua_State *L, int idx)
```

Expects a valid userdata metatable at the top of the stack, assigns it to the userdata residing at stack index idx, and pops the value at the top of the stack thereafter. If the value at the top of the stack is null, then a metatable assigned to a userdatum is deleted, and null is popped from the stack.

agn_setutype

```
void agn_setutype (lua_State *L, int idxobj, int idxtype);
```

Sets a user-defined type of a procedure, table, sequence, set, userdata, or pair. The object is at stack index `idxobj`, the type (a string) is at position `idxtype`. The function leaves the stack unchanged.

If **null** is at `idxtype`, the function deletes the user-defined type.

Setting the type of a sequence, set, table, procedure, or pair also causes the pretty printer to display the string passed to the function instead of the usual output at the console. See also: **agn_getutype**.

agn_size

```
int agn_size (lua_State *L, int idx);
```

Returns the number of items currently stored to the array and the hash part of the table at stack index `idx`. See also: **agn_asize**.

agn_ssize

```
int agn_ssize (lua_State *L, int idx);
```

Returns the number of items currently stored to the set at stack index `idx`.

agn_sstate

```
void agn_sstate (lua_State *L, int idx, size_t a[])
```

Returns the actual number of items and the current maximum number of items allocable to the set at index `idx` in `a`, a C array with three entries. The actual number of items is stored to a[0], the current allocable size to a[1]. a[2] indicates whether a metatable has been assigned to the set, where 0 means it does not, and 1 that it does.

agn_stralloc

```
char *agn_stralloc (lua_State *L, size_t l, const char *procname, ...);
```

Allocates a string buffer by internally determining its most efficient size, aligned along the "long" boundary. The return is a char* pointer to the beginning of the string. The function zeros only the last few bytes and assumes that the `trailing` rest will be filled by real characters later on. Just pass `l` as the number of characters, excluding the terminating \0, and do not multiply it by sizeof(char). The function automatically adds a terminating \0.

The function can optionally free variables passed after `procname` in case memory allocation fails internally. In case of an error, the function issues the name of the procedure `procname` from which it was called.

In any case, the last argument must always be NULL.

agn_strmatch

```
const char *agn_strmatch (lua_State *L, const char *s, size_t s_len,
    const char *p, ptrdiff_t init, ptrdiff_t *start, ptrdiff_t *end)
```

Searches string `s` of size `s_len` for pattern `p`. `init` is the position from where to start the search and by default is 1, the first character in `s`. `start` and `end` will include the start and end position in case of a match, always counting from 1. The return is the string starting at position `start` in case of a match, or NULL if there was no hit.

agn_structinsert

```
void agn_structinsert (lua_State *L, int idxs,int idxv)
```

Inserts the object at stack index `idxv` into the table, set, sequence or register at stack index `idxs`. The function does not change the stack.

agn_tablesize

```
void agn_tablesize (lua_State *L, int idx, size_t a[])
```

Returns a guess on the number of elements in a table at stack index `idx` in `a[0]`, an indicator on whether a table contains an allocated hash part `a[1]`, and an indicator on whether null has been assigned to a table (`a[2]`).

The function is useful to determine the size of a table much more quickly than the **size** operator does, using a logarithmic instead of linear method, but may return incorrect results if the array part of a table has holes, so the programmer should make sure that the array part of a table has no holes. It also does not count the number of elements in the hash part of a table.

See also: **agn_tablestate**.

agn_tablestate

```
void agn_tablestate (lua_State *L, int idx, size_t a[], int mode)
```

Returns the number of key~value pairs allocable and actually assigned to the respective array and hash sections of the table at index `idx` by storing the result in `a`, a C array with nine entries.

The number of key~value pairs currently stored in the array part is stored to a[0], the number of pairs currently stored in the hash part to a[1]. a[2] contains the information whether the array part has holes (1) or not (0). The number of allocable key~value pairs to the array part is stored to a[3], and the number of allocable key~value pairs to the hash part is stored to a[4]. a[5] indicates whether **null** has been set to the table, where 0 = false, and 1 = true. If a[6] is 0, then the table does

not feature a metatable, if it is 1 then a metatable has been assigned. a[7] contains information on whether the hash part of a table does not have an allocated node (no dummynode), a[8] contains a guess on the number of elements in the array part of a table (see **agn_tablesize** for further information). a[9] contains the smallest integral index, and a[10] the largest integral index of the table.

If mode is not 1, then the number of pairs actually assigned is not determined, which may save time. In this case a[0] = a[1] = a[2] = 0.

agn_tabresize

```
LUA_API void agn_tabresize (lua_State *L, int idx, size_t newsize,
  int checkholes)
```

Resizes the array part of a table at stack index `idx` to the given number `newsize` of pre-allocated slots.

If you do not know whether there are still non-null values beyond the new size in the array part, then set `checkholes` to 1 instead of 0 as otherwise Agena will crash in such situations. If set to 1, the function will not resize the table if there are any excess non-null values.

agn_tocomplex (non-ANSI versions only)

```
agn_Complex agn_tocomplex (lua_State *L, int idx)
```

Assumes that the value at stack index `idx` is a complex value and returns it as a lua_Number. It does not check whether the value is a complex number.

agn_tointeger

```
lua_Integer agn_tointeger (lua_State *L, int idx)
```

Assumes that the value at stack index `idx` is a number and returns it as an integer, not a float. It does not check whether the value is a number.

The function does not change the stack.

agn_tonumber

```
lua_Number agn_tonumber (lua_State *L, int idx)
```

Assumes that the value at stack index `idx` is a number and returns it as a lua_Number. It does not check whether the value is a number. The values **undefined** and **infinity** are recognised properly.

The function does not change the stack.

agn_tonumberx

```
lua_Number agn_tonumberx (lua_State *L, int idx, int *exception)
```

If the value at stack index `idx` is a number or a string containing a number, it returns it as a lua_Number. The strings or names `'undefined'` and `'infinity'` are recognised properly. If successful, `exception` is assigned to 0.

If the value could not be converted to a number, 0 will be returned, and `exception` is assigned to 1.

agn_tostring

```
const char *agn_tostring (lua_State *L, int idx)
```

Assumes that the value at stack index `idx` is an Agena string and returns it as a C string of type const char *. It does not check whether the value is a string.

If `idx` is negative: due to garbage collection, there is no guarantee that the pointer returned will be valid after the corresponding value is removed from the stack.

agn_usedbytes

```
LUAI_UMEM agn_usedbytes (lua_State *L)
```

Returns the number of bytes used by the interpreter.

agnL_checkoption

```
LUALIB_API int agnL_checkoption (lua_State *L, int idx, const char *def,
                                 const char *const lst[], int ignorecase)
```

Like **luaL_checkoption**, but returns an error if a given option at index `idx` is not a string. Returns the index of the option found in `lst`, which may be the position of the default `def` if there is no value at idx. Otherwise issues an error if the option at `idx` is not part of `lst[]`, or if `def` is not in `lst[]`. `def` must not be NULL. If `ignorecase` is 1, the function compares option names case-insensitively, and case-sensitively if it is 0. Example:

```
static const char *const datatypes[] = {"uchar", "double", "int32", NULL};
position = agnL_checkoption(L, 2, "double", datatypes, 0);
```

agnL_createpairofnumbers

```
void agnL_createpairofnumber (lua_State *L, lua_Number l, lua_Number r);
```

Deprecated, see **agn_createpairnumbers**.

agnL_datetosecs

```
LUA_API Time64_T agnL_datetosecs (lua_State *L, int idx,
   const char *procname)
```

Takes either

- a table, register, or a sequence of date time values of the form [yy, mm, dd, hh, mm, seconds], or

- six numbers yy, mm, dd, hh, mm, seconds,

and returns the number of seconds elapsed since the begin of the epoch (usually January 01, 1970) as a Time64_t value; idx must be a _positive_ index number.

agnL_fncall

```
lua_Number agnL_fncall (lua_State *L, int idx, lua_Number x,
                        int optstart, int optstop);
```

Pushes the mathematical function at index `idx` and number `x` onto the stack, optionally pushes the numbers at stack positions `optstart` through `optstop` (including) and then calls the function with the values pushed. It always returns the first result of the function call.

The function does not change the stack and reserves its own stack space, so you do not have to call **lua_checkstack** or **luaL_checkstack**.

The function at `idx` should return one number, otherwise an error will be issued. If the function at `idx` is not multivariate, then pass values for `optstart` and `optstop` such that `optstart` > `optstop`. See also: agnL_fnunicall.

agnL_fneps

```
lua_Number agnL_fneps (lua_State *L, int fidx, lua_Number x, int n, int p,
int q, lua_Number *origh, lua_Number *abserr);
```

Determines an epsilon value by taking the function value f(x) into account, using a divided difference table. Also returns original epsilon estimate before correction in parameter `origh`, and the absolute error in parameter `abserr`.

The function must be at index position `idx`. `n` is the number of iterations and must be positive. `p` is the first index of further arguments to f, `q` the last index of further arguments to f. if `p` > `q`, no further arguments are evaluated.

agnL_fnunicall

```
lua_Number agnL_fnunicall (lua_State *L, int idx, lua_Number x);
```

Pushes the mathematical function at index `idx` and number `x` onto the stack, and then calls the function. It returns one result and leaves the stack unchanged.

The function reserves its own stack space, so you do not have to call **lua_checkstack** or **luaL_checkstack**.

See also: agnL_fncall.

agnL_geti

```
int agnL_geti (lua_State *L, int idx, int i);
```

Returns the `i`'th entry in the string, table array, pair, register, sequence or numarray at index `idx` and pushes it onto the top of the stack. The index `i` counts from 1.

agnL_getmetafield

```
agnL_getmetafield (lua_State *L, int idx, const char *event);
```

Pushes onto the stack the metamethod `event` from the metatable of the object at index `idx`. If the object does not have a metatable, or if the metatable does not have this field, or - unlike **luaL_getmetafield** - if the field does not refer to a procedure, returns 0 and pushes nothing. Examples for `event` are the metamethods `'__index'`, `'__tostring'`, etc.

agnL_getsetting

```
agnL_getsetting (lua_State *L, int idx,
  const char *const *modenames, const int *mode, const char *procname);
```

For a given string in `modenames` at stack index `idx`, returns the respective integral representation in `mode`. If the string passed cannot be found in `modenames`, issues an error.

Example:

```
static const int mode[] =
  {OP_ADD, OP_SUB, OP_MUL, OP_DIV, OP_INTDIV, OP_MOD, OP_POW, OP_IPOW};

static const char *const modenames[] =
  {"+", "-", "*", "/", "\\", "%", "^", "**", NULL};
```

```
int op = agnL_getsetting(L, 1, modenames, mode, "zip");
```

agnL_gettablefield

```
agnL_gettablefield (lua_State *L, const char *table, const char *field,
   const char *procname, int issueerror);
```

Determines the entry from the table field `<table>`.`<field>` and puts it on top of the stack. `procname` is the name of the function that calls **agnL_gettablefield**.

If `issueerror` is set to 1, then an error will be issued if `table` is not a table. If `issueerror` is set to 0 and `table` is not a table, then no such error will be issued and the global value found is pushed on the stack. In the latter case, the function returns LUA_TNONE-1.

The function returns the Lua/Agena type, an integer (e.g. LUA_TBOOLEAN), in case of success. If the `field` does not exist, LUA_TNIL will be returned and the function instead pushes **null** on top of the stack. See the agena.h source file for the proper type mapping (grep "basic types").

A typical call might look like this:

```
type = agnL_gettablefield(L, "environ", "infolevel",
  "environ.userinfo", 1);

if (type != LUA_TTABLE) {
  /* do something */
}
```

agnL_gettop

```
LUALIB_API  agnL_gettop  (lua_State  *L,  const  char  *message,  const  char
*procname);
```

Returns the number of arguments passed to a function, or issues an error with the given `message` if no argument has been passed.

agnL_isdlong

```
LUALIB_API int agnL_isdlong (lua_State *L, int idx)
```

Checks whether the value at stack index `idx` is a longdouble (see the **long** package) and returns 1 if it is and 0 otherwise.

agnL_optboolean

```
LUALIB_API int agnL_optboolean (lua_State *L, int narg, int def)
```

If the value at stack index `narg` is a Boolean, returns this Boolean as an integer: -1 for **fail**, 0 for **false**, and 1 for **true**. If there is no value at index `narg` or if it is **null**, returns `def`. Otherwise, raises an error.

agnL_optinteger

```
lua_Integer agnL_optinteger (lua_State *L, int narg, lua_Integer def)
```

If the function argument `narg` is a number, returns this number cast to a lua_Integer. If this argument is absent or is NULL, returns `def`. Otherwise, raises an error.

The function internally uses **agn_checknumber** which avoids internal calls to other C API auxiliary library functions and thus is somewhat faster than **luaL_optinteger**.

See also: **agnL_optposint**, **agnL_optnonnegint**, **agnL_optuint32_t**.

agnL_optnonnegative

```
lua_Number agnL_optnonnegative (lua_State *L, int narg, lua_Integer def)
```

If the function argument `narg` is a non-negative number (integer or float), returns this number cast to a lua_Number. If this argument is absent or is NULL, returns `def`. Otherwise, raises an error.

See also: **agnL_optpositive**.

agnL_optnonnegint

```
lua_Integer agnL_optnonnegint (lua_State *L, int narg, lua_Integer def)
```

If the function argument `narg` is a positive integer or zero, returns this number cast to a lua_Integer. If this argument is absent or is NULL, returns `def`. Otherwise, raises an error.

See also: **agnL_optinteger**, **agnL_optposint**.

agnL_optnumber

```
LUALIB_API agnL_optnumber(lua_State *L, int narg, lua_Number d)
```

If the value at stack index `narg` is a number, returns this number. If this stack value is absent or is NULL, returns `d`. Otherwise, raises an error. Contrary to **luaL_optnumber**, **agnL_optnumber** does not try to convert a string to a number.

agnL_optposint

```
lua_Integer agnL_optposint (lua_State *L, int narg, lua_Integer def)
```

If the function argument `narg` is a positive integer, returns this number cast to a lua_Integer. If this argument is absent or is NULL, returns `def`. Otherwise, raises an error.

See also: **agnL_optinteger**, **agnL_optnonnegint**, **agnL_optuint32_t**.

agnL_optpositive

```
lua_Number agnL_optpositive (lua_State *L, int narg, lua_Integer def)
```

If the function argument `narg` is a positive number (integer or float), returns this number cast to a lua_Number. If this argument is absent or is NULL, returns `def`. Otherwise, raises an error.

See also: **agnL_optnonnegative**, **agnL_optuint32_t**.

agnL_optstring

```
LUALIB_API const char *agnL_optstring (lua_State *L, int narg,
                                       const char *def)
```

Similar to **luaL_optstring**, but returns an error if a given option is not a string. The length of the optional string is not determined.

agnL_optuint32_t

```
lua_Integer agnL_optuint32_t (lua_State *L, int narg, uint32_t def)
```

Checks for an optional argument at stack position `narg` and returns it if it is a non-negative number; if the given argument is not a non-positive number, issues an error. If `narg` is null, returns `def`.

See also: **agn_checkuint32_t**, **agnL_optinteger**, **agnL_optnonnegint**, **agnL_optposint**.

agnL_paircheckbooloption

```
agnL_paircheckbooloption (lua_State *L, const char *procname, int idx,
    const char *option)
```

For the given Agena procedure `procname`, checks whether the value at index i_{dx} is a pair, and whether its left operand is equals to `option` (of type string), and whether the right operand is a Boolean.

Returns -2 if the value at `idx` is not a pair, or the result of the call to the lua_toboolean C API function.

The function issues an error if the left operand of the pair is not equals to `option`, or if the right operand is not a Boolean.

The function does *not* pop the pair at `idx`.

agnL_pairgetinumber

```
lua_Number agnL_pairgetinumber (lua_State *L, const char *procname,
int idx, int place)
```

For the given Agena procedure `procname`, checks whether the value at *stack index* i_{dx} is a pair. It then checks whether the left-hand (pos=1) or right-hand side (pos=2) is annumber and returns itthese numbers in `x` and `y`. The function leaves the stack unchanged otherwise.

If the value at i_{dx} is not a pair, or if at least one of its operands is not a number, it issues an error.

agnL_pairgetinumbers

```
void agnL_pairgetinumbers (lua_State *L, const char *procname, int idx,
    lua_Number *x, lua_Number *y)
```

For the given Agena procedure `procname`, checks whether the value at *stack index* i_{dx} is a pair. It then checks whether the left-hand and right-hand side are numbers and returns these numbers in `x` and `y`. Finally, if `idx` is negative, the function pops the pair from the stack, and leaves the stack unchanged otherwise.

If the value at i_{dx} is not a pair, or if at least one of its operands is not a number, it issues an error.

agnL_pexecute

```
int agnL_pexecute (lua_State *L, const char *str, const char *procname)
```

Executes the operating system command represented by string `str` and puts the output - a string - onto the top of the stack.

The string `procname` indicates the name of the function from which it is called. The function also removes any carriage returns(`'\r'`) from the output.

agnL_readlines

```
void agnL_readlines (lua_State *L, FILE *f,
   const char *procname, int ispipe)
```

Reads in the file or pipe depicted by its handle `f` and pushes its entire contents or output as a string onto the top of the stack. The string `procname` indicates the name of the function from which it is called. The function also removes any carriage returns (`'\r'`) from the output.

If you read from a pipe, pass 1 for `ispipe`, and 0 otherwise.

Note that you have to open the file or pipe before, and that the function does not close the file or pipe automatically.

agnL_tonumarray

```
lua_Number *agnL_tonumarray (lua_State *L, int idx, size_t *size,
                            const char *procname)
```

Creates a C double array and puts all numbers in the table, sequence or register at index `idx` into it. If a non-number is part of the structure, an error will be issued for function `procname`. `size` will contain the number of elements in the C array after returning. Free the array after usage. See also **agnL_fillarray**.

agnL_tostringx

```
const char *(agnL_tostringx) (lua_State *L, int idx, const char *procname);
```

Converts the number, Boolean, **null** or string at index `idx` to a string. If a wrong argument has been passed, the function issues an error with the name of the procedure causing the exception given by `procname`.

lua_absindex

```
int lua_absindex (lua_State *L, int idx);
```

Converts the acceptable index `idx` into an equivalent absolute index (that is, one that does not depend on the stack top).

lua_arith

```
void lua_arith (lua_State *L, int op);
```

Performs an arithmetic operation `op` over the two values (or one, in the case of negations) at the top of the stack, with the value at the top being the second operand, pops these values, and pushes the result of the operation. The function follows the semantics of the corresponding Agena operator (that is, it may call metamethods).

The value of `op` must be one of the following constants:

- OP_ADD: performs addition (+)
- OP_SUB: performs subtraction (-)
- OP_MUL: performs multiplication (*)
- OP_DIV: performs float division (/)
- OP_INTDIV: performs integer division (\)
- OP_MOD: performs modulo (%)
- OP_POW: performs exponentiation (^)
- OP_IPOW: performs integer exponentiation (**)
- OP_UNM: performs mathematical negation (unary -)

See also: **lua_compare**.

lua_compare

```
void lua_compare (lua_State *L, int idx1, int idx2, int op);
```

Compares two Agena values. Returns 1 if the value at index `idx1` satisfies `op` when compared with the value at index `idx2`, following the semantics of the corresponding Agena operator (that is, it may call metamethods). Otherwise returns 0. Also returns 0 if any of the indices is not valid.
The value of `op` must be one of the following constants:

- OP_EQ: compares for equality (==),
- OP_LT: compares for less than (<),
- OP_LE: compares for less or equal (<=).

See also: **lua_arith**.

lua_copy

```
void lua_copy (lua_State *L, int fromidx, int toidx);
```

Copies the element at index `fromidx` into the valid index `toidx`, replacing the value at that position. Values at other positions are not affected.

lua_geti

```
int lua_geti (lua_State *L, int idx, lua_Integer i);
```

Pushes onto the stack the value t[`i`], where t is the value at the given index. This function may trigger a metamethod for the '`__index`' event. Returns the type of the pushed value. See also: **lua_seti**.

lua_getiuservalue

```
void lua_getiuservalue (lua_State *L, int idx, int n);
```

Pushes onto the stack the `n`-th user value associated with the full userdata at the given index and returns the type of the pushed value.

If the userdata does not have that value or the value at `idx` is not a userdata, pushes **null** and returns LUA_TNONE.

`n` should always be 1 as the function would always return LUA_TNONE otherwise.

lua_getwarnf

```
int lua_setwarnf (lua_State *L);
```

Returns the current warning state:

- 0 - warning system is off;
- 1 - ready to start a new message;
- 2 - previous message is to be continued.

See also: **lua_setwarnf**.

lua_hasfield

```
void lua_hasfield (lua_State *L, int idx, const char *k);
```

Checks whether a table at index `idx` includes a given field, a string `key`, and returns 0 (false) or 1 (true). Metamethods are ignored. The stack is left unchanged.

See also: **lua_getfield**, **lua_shas**.

lua_iscomplex

```
void lua_iscomplex (lua_State *L, int idx);
```

This macro checks whether the value at stack index `idx` is a complex number. It returns 1 if the value is a complex number, and 0 otherwise. It does *not* pop anything.

lua_newuserdatuv

```
void *lua_newuserdatauv (lua_State *L, size_t size, int nuvalue);
```

This function creates and pushes on the stack a new full userdata, with `nuvalue` associated Agena values, called user values, plus an associated block of raw memory with size bytes. (The user values can be set and read with the functions **lua_setiuservalue** and **lua_getiuservalue**.) In Agena, `nuvalue` should always be 1, otherwise the function will issue an error to avoid segmentation faults.

The function returns the address of the block of memory.

lua_isnone

```
void lua_isnone (lua_State *L, int idx);
```

This macro returns 1 if the given index is acceptable, but not valid, and 0 otherwise.

lua_isreg

```
void lua_isreg (lua_State *L, int idx);
```

This macro checks whether the value at stack index `idx` is a register. It returns 1 if the value is a pair, and 0 otherwise. It does *not* pop anything.

lua_ispair

```
void lua_ispair (lua_State *L, int idx);
```

This macro checks whether the value at stack index `idx` is a pair. It returns 1 if the value is a pair, and 0 otherwise. It does *not* pop anything.

lua_isseq

```
void lua_isseq (lua_State *L, int idx);
```

This macro checks whether the value at stack index `idx` is a sequence. It returns 1 if the value is a sequence, and 0 otherwise. It does *not* pop anything.

lua_isset

```
void lua_isset (lua_State *L, int idx);
```

This macro checks whether the value at stack index `idx` is a set. It returns 1 if the value is a set, and 0 otherwise. It does *not* pop anything.

lua_isyieldable

```
void lua_isyieldable (lua_State *L);
```

This macro returns 1 if the given coroutine can yield, and 0 otherwise.

lua_numbertointeger

```
int lua_numbertointeger (lua_Number n, lua_Integer *p);
```

Tries to convert a float to an integer; the floating-point value `n` must have an integral value. If that value is within the range of integers, it is converted to an integer and assigned to `*p`. The macro results in a boolean indicating whether the conversion was successful. (Note that this range test can be tricky to do correctly without this macro, due to rounding.)

This macro may evaluate its arguments more than once.

lua_pushchar

```
void lua_pushchar (lua_State *L, char c);
```

Pushes character `c` as a string of length 1 onto the stack.

lua_pushfail

```
void lua_pushfail (lua_State *L);
```

This macro pushes the Boolean value **fail** onto the stack.

lua_pushfalse

```
void lua_pushfalse (lua_State *L);
```

This macro pushes the Boolean value **false** onto the stack.

lua_pushglobaltable

```
void lua_pushglobaltable (lua_State *L);
```

Pushes the global environment onto the stack.

lua_pushundefined

```
void lua_pushundefined (lua_State *L);
```

Pushes the value **undefined** onto the stack.

lua_pushunsigned

```
LUA_API void lua_pushunsigned (lua_State *L, lua_Unsigned u);
```

Pushes an unsigned int (if you do not change the defaults) onto the stack.

lua_pushtrue

```
void lua_pushtrue (lua_State *L);
```

This macro pushes the Boolean value **true** onto the stack.

lua_rawaequal

```
int lua_rawaequal (lua_State *L, int index1, int index2);
```

Returns 1 if the two values in acceptable indices `index1` and `index2` are primitively approximately equal (that is, without calling metamethods, see also **approx**, ~=). Otherwise returns 0. Also returns 0 if any of the indices are non valid.

lua_rawgetp

```
int lua_rawgetp (lua_State *L, int index, const void *p);
```

Pushes onto the stack the value t[k], where t is the table at the given index and k is the pointer `p` represented as a light userdata. The access is raw; that is, it does not use the `__index` metavalue.

Returns the type of the pushed value.

lua_rawset2

```
void lua_rawset2 (lua_State *L, int idx);
```

Similar to **lua_settable**, but does a raw assignment (i.e., without metamethods).

Contrary to **lua_rawset**, only the value is deleted from the stack, the key is kept, thus you save one call to **lua_pop**. This makes it useful with **lua_next** which needs a key in order to iterate successfully.

lua_rawsetiboolean

```
void lua_rawsetiboolean (lua_State *L, int idx, int n, int num);
```

This macro does the equivalent of t[n] := `fail`, if num is -1, t[n] := `false`, if num is 0, and t[n] := `true`, if num is 1. t is the table at the given valid index `idx`, n is an integer, and `num` an integer.
This function leaves the stack unchanged. The assignment is raw; that is, it does not invoke metamethods.

lua_rawsetilstring

```
void lua_rawsetilstring (lua_State *L, int idx, int n, const char *str,
    int len);
```

This macro does the equivalent of t[n] := string, where t is the table at the given valid index `idx`, n is an integer, `str` the string to be inserted and `len` the length of then string.

This function leaves the stack unchanged. The assignment is raw; that is, it does not invoke metamethods.

lua_rawsetikey

```
void lua_rawsetikey (lua_State *L, int idx, int n);
```

Does the equivalent of t[n] := k, where t is the table at the given valid index `idx` and k is the value just below the top of the stack.

This function pops the topmost value from the stack and leaves everything else untouched. The assignment is raw; that is, it does not invoke metamethods.

lua_rawsetinumber

```
void lua_rawsetinumber (lua_State *L, int idx, int n, lua_Number num);
```

This macro does the equivalent of t[n] := `num`, where t is the table at the given valid index `idx`, n is an integer, and `num` an Agena number (a C double).

This function leaves the stack unchanged. The assignment is raw; that is, it does not invoke metamethods.

lua_rawsetistring

```
void lua_rawsetistring (lua_State *L, int idx, int n, const char *str);
```

This macro does the equivalent of t[n] = `str`, where t is the table at the given valid index `idx`, n is an integer, and `str` a string.

This function leaves the stack unchanged. The assignment is raw; that is, it does not invoke metamethods.

lua_rawsetp

```
void lua_rawsetp (lua_State *L, int index, const void *p);
```

Does the equivalent of t[p] = v, where t is the table at the given index, `p` is encoded as a light userdata, and v is the value on the top of the stack.

This function pops the value from the stack. The assignment is raw, that is, it does not use the `__newindex` metamethod.

lua_rawsetstringboolean

```
void lua_rawsetstringboolean
    (lua_State *L, int idx, const char *str, int n);
```

This macro does the equivalent of t[`str`] := (`n` == 1), where t is the value at the given valid index `idx`, `str` a string, and `n` an integer.

This function leaves the stack unchanged. The assignment is raw; that is, it does not invoke metamethods.

lua_rawsetstringchar

```
void lua_rawsetstringchar
    (lua_State *L, int idx, const char *str, int v);
```

This macro does the equivalent of t[str] := v, where t is the value at the given valid index idx, str a string, and v is an integer.

This function leaves the stack unchanged. The assignment is raw; that is, it does not invoke metamethods. See also **lua_rawsetstringstring**.

lua_rawsetstringnumber

```
void lua_rawsetstringnumber
    (lua_State *L, int idx, const char *str, lua_Number n);
```

This macro does the equivalent of t[str] := n, where t is the value at the given valid index idx, str a string, and n a number.

This function leaves the stack unchanged. The assignment is raw; that is, it does not invoke metamethods.

lua_rawsetstringpairnumbers

```
void lua_rawsetstringpairnumbers
    (lua_State *L, int idx, const char *str, lua_Number x, lua_Number y);
```

This macro does the equivalent of t[str] := x:y, where t is the value at the given valid index idx, str a string, and x:y is a pair of the numbers x and y.

This function leaves the stack unchanged. The assignment is raw; that is, it does not invoke metamethods.

lua_rawsetstringstring

```
void lua_rawsetstringstring
    (lua_State *L, int idx, const char *str, const char *text);
```

This macro does the equivalent of t[str] := text, where t is the value at the given valid index idx, str a string, and text is a string.

This function leaves the stack unchanged. The assignment is raw; that is, it does not invoke metamethods. See also **lua_rawsetstringchar**.

lua_reginsert

```
void lua_reginsert (lua_State *L, int idx);
```

Puts the element on top of the Lua stack into the register at stack index `idx`. on the first slot that is currently set to **null**. If there is no free slot, an error will be issued. The value added to the register is popped from the stack thereafter.

lua_regnext

```
int lua_regnext (lua_State *L, int idx);
```

Pops a key from the stack, and pushes the next key~value pair from the register at the given index `idx`. If there are no more elements in the register or the position of the top pointer has been exceeded, then **lua_regnext** returns 0 (and pushes nothing). To access the very first item in a register, put **null** on the stack before (with **lua_pushnil**).

While traversing a register, do not call **lua_tolstring** directly on the key. Recall that **lua_tolstring** changes the value at the given index; this confuses the next call to **lua_regnext**.

lua_regsetinumber

```
void lua_regsetinumber (lua_State *L, int idx, int n, lua_Number num);
```

This macro sets the given Agena number `num` to the non-zero and positive index `n` of the register at stack index `idx`.

lua_rotate

```
void lua_rotate (lua_State *L, int idx, int n);
```

Rotates the stack elements between the valid index `idx` and the top of the stack. The elements are rotated `n` positions in the direction of the top, for a positive `n`, or `-n` positions in the direction of the bottom, for a negative `n`.

The absolute value of n must not be greater than the size of the slice being rotated.

This function cannot be called with a pseudo-index, because a pseudo-index is not an actual stack position.

lua_sdelete

```
void lua_sdelete (lua_State *L, int idx);
```

Deletes the element residing at the top of the stack from the set at stack position `idx`. The element at the stack top is popped thereafter.

Note that you should not delete a value while traversing a set with lua_usnext. This code, however, works:

```
lua_pushvalue(L, idx);
if (lua_isset(L, -1)) {
  lua_pushnil(L);
  while (lua_usnext(L, -2) != 0) {
    lua_sdelete(L, -3);  /* delete value in the set and pop it */
    agn_poptop(L);       /* pop the key from stack, too, since it cannot
                            be used for next iteration as it has been
                            purged from the set */
    lua_pushnil(L);      /* restart iteration */
  }
}
agn_poptop(L);
```

See also: **lua_sinsert**.

lua_seqgeti

```
void lua_seqgeti (lua_State *L, int idx, int n);
```

Gets the `n`-th item from the sequence at stack index `idx` and pushes it onto the stack. You have to make sure that the index is valid, otherwise there may be segmentation faults.

See also: **lua_seqseti**.

lua_seqrawgetinumber

```
lua_Number lua_seqrawgetinumber (lua_State *L, int idx, int n);
```

Returns the value t[n] *as a* lua_Number, where t is a sequence at the given valid index `idx`. If t[n] is not a number, the return is `HUGE_VAL`. The access is raw; that is, it does not invoke metamethods.

See also: **agn_seqgetinumber**.

lua_seqinsert

```
void lua_seqinsert (lua_State *L, int idx);
```

Inserts the element on top of the Lua stack into the sequence at stack index `idx`. The element is inserted at the end of the sequence. The value added to the sequence is popped from the stack thereafter.

lua_seqnext

```
int lua_seqnext (lua_State *L, int idx);
```

Pops a key from the stack, and pushes the next key~value pair from the sequence at the given index `idx`. If there are no more elements in the sequence, then **lua_seqnext** returns 0 (and pushes nothing). To access the very first item in a sequence, put **null** on the stack before (with **lua_pushnil**).

While traversing a sequence, do not call **lua_tolstring** directly on the key. Recall that **lua_tolstring** changes the value at the given index; this confuses the next call to **lua_seqnext**.

lua_seqrawget

```
void lua_seqrawget (lua_State *L, int idx, int pushnil);
```

Pushes onto the stack the sequence value t[k], where t is the sequence at the given valid index `idx` and k is the value at the top of the stack. If t[k] does not exist, an error will be issued if `pushnil` = 0, and **null** will be pushed if `pushnil` is non-zero.

This function pops the key from the stack (putting the resulting value in its place). The function does not invoke any metamethods.

lua_seqrawgeti

```
void lua_seqrawgeti (lua_State *L, int idx, size_t n);
```

Pushes onto the stack the sequence value t[n], where t is the sequence at the given valid index `idx`.

The function does not invoke any metamethods. Contrary to **lua_rawgeti**, it issues an error if `n` is out of range.

lua_seqrawset

```
void lua_seqrawset (lua_State *L, int idx);
```

Does the equivalent to s[k] := v, where s is a sequence at the given valid index `idx`, v is the value at the top of the stack, and k is the value just below the top.

This function pops both the key and the value from the stack. It does not invoke any metamethods.

lua_seqrawsetilstring

```
void lua_seqrawsetilstring (lua_State *L, int idx, int n, const char *str,
    int len);
```

This macro does the equivalent of s[n] = `string`, where s is the sequence at the given valid index `idx`, n is an integer, `str` the string to be inserted and `len` the length of then string.

This function leaves the stack unchanged. The assignment is raw; that is, it does not invoke metamethods.

lua_seqseti

```
void lua_seqseti (lua_State *L, int idx, int n);
```

Sets the value at the top of the stack to the non-zero and positive index n of the sequence at stack index `idx`.

If the value added is **null**, the entry at sequence index n is deleted and all elements to the right of the value deleted are shifted to the left, so that their index positions get changed, as well.

The function pops the value from the top of the stack.

If there is already an item at position n in the sequence, it is overwritten.

If you want to extend a current sequence, the function allows to add a new item only at the next free index position. Larger index positions are ignored, but the value to be added is popped from the stack, as well.

See also: **lua_seqgeti**.

lua_seqsetinumber

```
void lua_seqsetinumber (lua_State *L, int idx, int n, lua_Number num);
```

The function sets the given Agena number `num` to the non-zero and positive index `n` of the sequence at stack index `idx`.

lua_seqsetistring

```
void lua_seqsetistring (lua_State *L, int idx, int n, const char *str);
```

This macro sets the given string `str` to the non-zero and positive index index `n` of the sequence at stack index `idx`.

lua_seti

```
void lua_seti (lua_State *L, int idx, lua_Integer n);
```

Does the equivalent to t[n] = v, where t is the value at the given index `idx` and v is the value at the top of the stack.

This function pops the value from the stack. This function may trigger a metamethod for the '`__writeindex`' event.

See also: **lua_geti**.

lua_setmetatabletoobject

```
void lua_setmetatabletoobject (lua_State *L, int idx, const char *k,
  int settype);
```

Sets metatable `k` to the structure at index position `idx`. The function does not change the stack. If `settype` is 1 the user-defined type `k` will also be set, if `settype` is 0 no user-defined type will be set. If `k` is NULL, the metatable will be deleted and if `settype` is 1, the user-defined type will also be removed.

lua_setwarnf

```
void lua_setwarnf (lua_State *L, lua_WarnFunction f, void *ud);
```

Sets the warning function to be used by Lua to emit warnings (see lua_WarnFunction). The `ud` parameter sets the value `ud` passed to the warning function. See also: **lua_getwarnf**.

lua_sinsert

```
void lua_sinsert (lua_State *L, int idx);
```

This macro to **lua_srawset** inserts an item into a set. The set is at the given index `idx`, and the item is at the top of the stack.

This function pops the item from the stack. See also: **lua_sdelete**.

lua_sinsertlstring

```
void lua_sinsertlstring (lua_State *L, int idx, const char *str, size_t l);
```

This macro sets the first l characters of the string denoted by `str` into the set at the given index `idx`.

lua_sinsertnumber

```
void lua_sinsertnumber (lua_State *L, int idx, lua_Number n);
```

This macro sets the number denoted by `n` into the set at the given index `idx`.

lua_shas

```
void lua_shas (lua_State *L, int idx, int pop);
```

Checks whether the value at the stack top exists in the set at stack index `idx`. Returns 0 or 1. If `pop` is 1, pops the value, otherwise leaves the stack unchanged.

See also **lua_hasfield**.

lua_sinsertstring

```
void lua_sinsertstring (lua_State *L, int idx, const char *str);
```

This macro sets the string denoted by `str` into the set at the given index `idx`.

lua_srawget

```
int lua_srawget (lua_State *L, int idx);
```

Checks whether the set at index `idx` contains the value at the top of the stack. The function pops the value from the stack putting the Boolean value **true** or **false** in its place. It returns 1 if the element has been found, and 0 otherwise.

It does not invoke any metamethods.

lua_srawset

```
void lua_srawset (lua_State *L, int idx);
```

Does the equivalent to `insert` v `into` s, where s is the set at the given valid index `idx`, v is the value at the top of the stack.

This function pops the value from the stack. It does not invoke any metamethods. To delete entries, see **lua_sdelete**.

lua_stringtonumber

```
void lua_stringtonumber (lua_State *L, const char *s);
```

Converts the zero-terminated string `s` to a number, pushes that number into the stack, and returns the total size of the string, that is, its length plus one. The conversion can result in an integer or a float, according to the lexical conventions of Agena. The string may have leading and trailing whitespaces and a sign. If the string is not a valid numeral, returns 0 and pushes nothing. (Note that the result can be used as a Boolean, true if the conversion succeeds.)

lua_toboolean

```
int lua_toboolean (lua_State *L, int idx)
```

Converts the value at the given acceptable index to an integer value (-1, 0 or 1).

If the value at `idx` is **null** or **false**, the functions returns 0.
If the value at `idx` is **fail**, the function returns -1.

If the value at `idx` is different from **false**, **fail**, and **null**, the function returns 1.

The function also returns 0 when called with a non-valid index. (If you want to accept only actual Boolean values, use **lua_isboolean** to test the value's type.)

lua_toint32_t

```
int32_t lua_toint32_t (lua_State *L, int idx)
```

Converts the value at the given acceptable index to the signed integral type int32_t. The value must be a number or a string convertible to a number; otherwise, **lua_toint32_t** returns 0.

If the number is not an integer, it is truncated in some non-specified way.

See also: **agnL_optuint32_t**.

lua_usnext

```
int lua_usnext (lua_State *L, int idx);
```

Pops a key from the stack, and pushes the next item twice (!) from the set at the given `idx`. If there are no more elements in the set, then **lua_usnext** returns 0 (and pushes nothing). To access the very first item in a set, put **null** on the stack before (with **lua_pushnil**).

While traversing a set, do not call **lua_tolstring** directly on an item, unless you know that the item is actually a string. Recall that **lua_tolstring** changes the value at the given index; this confuses the next call to **lua_usnext**.

lua_warning

```
int lua_warning (lua_State *L, const char *msg, int tocont);
```

Emits a warning with the given message. A message in a call with `tocont` true should be continued in another call to this function.

luaL_addgsub

```
const void luaL_addgsub (luaL_Buffer *B, const char *s,
                         const char *p, const char *r);
```

In string `s`, replaces any occurrence of the string `p` with the string `r` and adds the resulting string to buffer `B`. Search patterns are ignored.

luaL_argexpected

```
void luaL_argexpected (lua_State *L, int cond, int arg, const char *tname);
```

Checks whether `cond` is true. If it is not, raises an error about the type of the argument `arg` with a standard message (see **luaL_typeerror**).

luaL_checkint32_t

```
int32_t luaL_checkint32_t (lua_state *L, int narg)
```

Checks whether the function argument `narg` is a number and returns this number cast to an int32_t.

luaL_checksetting

```
int luaL_checksetting (lua_State *L, int idx,
  const char *const lst[], const char *errmsg);
```

Checks whether the string at stack index `idx` is included in the list `lst`, and returns its position in `lst`, counting from 0. If it does not find the string in the list, issues the error `errmsg`.

luaL_clearbuffer

```
void luaL_clearbuffer (luaL_Buffer *B)
```

Clears a luaL_Buffer and resets it, does not leave anything on the stack

luaL_isudata

```
int luaL_isudata (lua_State *L, int ud, const char *tname);
```

Checks whether the object at stack position `ud` is a userdata object. If `ud` depicts the position of a userdata, the function also checks whether the metatable - if available - attached to it complies with metatable `tname` which the programmer originally intended to be used by the specific userdata.

The function returns 1 if all checks have been successful, and 0 otherwise.

See also: **luaL_getudata**.

luaL_getsubtable

```
int luaL_getsubtable (lua_State *L, int idx, const char *fname);
```

Ensures that the value t[fname], where t is the value at index idx, is a table, and pushes that table onto the stack. Otherwise creates a new table, assigned to t[fname], and pushes it onto the top of the stack. Returns true (1) if it finds a previous table and false (0) if it creates a new table.

luaL_getudata

```
void *luaL_getudata (lua_State *L, int narg, const char *tname,
                     int *result);
```

Checks whether the function argument narg is a userdata of the type tname. Contrary to **luaL_checkudata**, it does not issue an error if the argument is not a userdata, and also stores 1 to result if the check was successful, and 0 otherwise.

luaL_setfuncs

```
void luaL_setfuncs (lua_State *L, const luaL_Reg *l, int nup);
```

Registers all functions in the array l (see luaL_Reg) into the table on the top of the stack (below optional upvalues, see next).

When nup is not zero, all functions are created with nup upvalues, initialized with copies of the nup values previously pushed on the stack on top of the library table. These values are popped from the stack after the registration.

The following functions have originally been written by Rici Lake for Lua 5.x:

luaL_newref

```
luaRef *luaL_newref (lua_State *L, int idx);
```

Creates a new C reference to the object at stack index `idx`.

luaL_pushref

```
void luaL_pushref (lua_State *L, luaRef *r)
```

Pushes a referenced object onto the stack.

luaL_pushref

```
void luaL_freeref (lua_State *L, luaRef *r)
```

Frees a reference.

Appendices

Appendix A

A1 Operators

Unary operators are:

```
&&, ~~, ||, ^^, abs, antilog2, antilog10, arccos, arcsec, arcsin, arctan, assigned,
atendof, bea, char, cis, conjugate, copy, cos, cosh, cosxx, cube, empty, entier, even,
exp, filled, finite, first, flip, float, lngamma, gethigh, getlow, imag, infinite,
inrange, instr, int, integral, join, last, left, ln, lower, nan, nargs, nonzero, not,
odd, qmdev, qsumup, real, recip, reg, replace, right, sumup, seq, sign, signum, sin,
sinc, sinh, size, square, sqrt, tan, tanh, trim, type, unassigned, unique, upper,
times, typeof, values, zero, -
```
(unary minus), `~~` (bitwise complement).

Binary operators are:

`and`, `in`, `intersect`, `minus`, `nand`, `nor`, `or`, `roll`, `split`, `squareadd`, `subset`, `symmod`, `union`, `xor`, `xnor`, `xsubset`, `+` (addition), `-` (subtraction), `*` (multiplication), `/` (division), `*%` (percentage) `/%` (ratio), `+%` (add percentage), `-%` (subtract percentage), `%%` (percentage change), `\` (integer division), `%` (modulus), `^` (exponentiation), `**` (integer exponentiation), `&` (concatenation), `=` (equality), `~=` (approximate equality), `~<>` (approximate inequality), `<` (less than), `<=` (less or equal), `>` (greater than), `>=` (greater or equal), `@` (mapping), `$` (selection), `$$` (fulfillment), `:` (pair constructor), `!` (complex constructor), `&&` (bitwise and), `||` (bitwise or), `^^` (bitwise xor), `<<<` (bitwise left-shift), `>>>` (right-shift), `<<<<` (bitwise left-rotation), `>>>>` (right-rotation), `&+` (add 4-byte integer), `&-` (subtract 4-byte integer), `&*` (multiply 4-byte integer), and `&/` (devide 4-byte integer), `|` (comparison), `~|` (approximate comparison), `|-` (absolute difference).

A2 Metamethods

The following metamethods were inherited from Lua 5.1:

Index to metatable	Meaning
'__index'	read operation using indices, e.g. if n[1] = 0 then ... or if n[2 to 3] = 'JP' then ...
'__gc'	Garbage collection, for userdata only
'__mode'	Sets weakness of a table
'__add'	Addition of two values
'__sub'	Subtraction of two values
'__mul'	Multiplication of two values
'__div'	Division of two values
'__mod'	Modulus
'__pow'	Exponentiation
'__unm'	Unary minus
'__eq'	Equality operation
'__lt'	Less-than operation
'__le'	Less-than or equals operation
'__concat'	Concatenation
'__call'	See Lua 5.1 manual
'__tostring'	Method for pretty printing values at stdout
'__metatable'	Protection for metatables
'__weak'	Declaration of weak tables, sets, and sequences

Table 20: Metamethods taken from Lua

The **__len** metamethod in Lua 5.1 to determine the size of an object was replaced with the **__size** metamethod. Lua's **__mode** metamethod has been renamed **__weak**.

The following methods are new in Agena:

Index to metatable	Meaning
'__abs'	**abs** operator
'__aeq'	approximate equality operator ($\sim=$)
'__arccos'	**arccos** operator
'__arcsin'	**arcsin** operator
'__arctan'	**arctan** operator
'__cis'	**cis** operator
'__cos'	**cos** operator
'__cosh'	**cosh** operator
'__cube'	**cube** operator
'__eeq'	strict equality operator (==)
'__empty'	**empty** operator
'__even'	**even** operator
'__exp'	**exp** operator
'__filled'	**filled** operator
'__float'	**float** operator

Index to metatable	Meaning
`'__imag'`	**imag** operator
`'__in'`	**in** binary operator
`'__intdiv'`	integer division
`'__intersect'`	**intersect** operator
`'__integral'`	**integral** operator
`'__ipow'`	exponentiation with an integer power
`'__ln'`	**ln** operator
`'__log'`	**log** operator
`'__minus'`	**minus** operator
`'__nonzero'`	**nonzero** operator
`'__notin'`	**notin** operator
`'__oftype'`	self-defined type check for **::**, **:-**, and parameter lists of procedures
`'__qsumup'`	**qsumup** operator for table or sequence-based user-defined types
`'__real'`	**real** operator
`'__recip'`	**recip** operator
`'__sumup'`	**sumup** operator for table or sequence based user-defined types
`'__sign'`	**sign** operator
`'__sin'`	**sin** operator
`'__sinc'`	**sinc** operator
`'__sinh'`	**sinh** operator
`'__size'`	**size** operator
`'__sqrt'`	**sqrt** operator
`'__square'`	**square** operator
`'__squareadd'`	**squareadd** operator
`'__tan'`	**tan** operator
`'__tanh'`	**tanh** operator
`'__union'`	**union** operator
`'__writeindex'`	write operation using indices, e.g. n[1] := 0
`'__zero'`	**zero** operator

Table 21: Metamethods introduced with Agena

A3 Mathematical Constants

Constant	Meaning
degrees	Factor $1/\pi*180$ to convert radians to degrees
DoubleEps	Equals 2.2204460492503131E-16
Eps	Equals 1.4901161193847656e-08
hEps	Equals 1.4901161193847656e-12
EulerGamma	Euler-Mascheroni constant, equals 0.577215664901532860061
E, **Exp**	Constant e = exp(1) = 2.71828182845904523536
I	Imaginary unit $\sqrt{-1}$
infinity	Infinity ∞
Pi	Constant π = 3.14159265358979323846
Pi2	Constant 2π = 6.28318530717958646926
PiO2	Constant $\pi/2$ = 1.570796326794896619232
PiO4	Constant $\pi/4$ = 0.785398163397448309616
radians	Factor $\pi/180$ to convert degrees to radians
undefined	An expression stating that it is undefined, e.g. a singularity
Phi	Golden ratio $(1+\sqrt{5})/2$
math.largest	Largest representable number; the smallest negative one nearest to $-\infty$ is the negative of this constant
math.smallest	Smallest positive representable number
math.smallest-normal	Smallest positive normal number
math.lastcontint	Largest integer i representable on the floating-point system with enough precision, such that i - 1 <> i
math.exa	10^{18}
math.peta	10^{15}
math.tera	10^{12}
math.giga	10^{9}
math.mega	10^{6}
math.kilo	10^{3}
math.deka	10^{1}
math.deci	10^{-1}
math.centi	10^{-2}
math.milli	10^{-3}
math.micro	10^{-6}
math.nano	10^{-9}
math.pico	10^{-12}
math.femto	10^{-15}
math.atto	10^{-18}

Table 27: Constants

A4 System Variables

Agena lets you configure the following settings, where `n/e` means `no effect`.

System variable	Meaning	Write
libname	The paths to Agena libraries.	yes
mainlibname	The path to the main Agena directory.	yes
environ.cpu	Contains the name of the CPU in use as a lower-case string, e.g. `'sparc'`, `'ppc'` for PowerPC, or `'x86'` for Intel 386-compatible processors. See also system variable **environ.os**.	no
environ.homedir	The path to the user's home directory.	yes
environ.gdidefaultoptions	A table with all default plotting options for some functions in the **gdi** package. This table is set by **gdi.setoptions**.	no
environ.libpatchlevel	The update version of the main Agena library (in lib/library.agn). Mostly defaults to **null**.	no
environ.maxpathlength	The maximum number of characters for a file path (excluding C's \0 character).	no
environ.more	The number of entries in tables and sets printed by **print** and the end-colon functionality before issuing the `press any key` prompt. Default is 40.	yes
environ.os	Contains the name of the operating system in use as a lower-case string, e.g. `'windows'`, `'macosx'`, `'solaris'`, `'os/2'`, `'haiku'`, `'dos'`, or `'linux'`. Do not change this value. See also system variable **environ.cpu**.	no
environ.release	A sequence containing the string `AGENA`, the main interpreter version as a number, the subversion as a number, and the C patch number as a number, as well. The lib/library.agn patch level is denoted by the fourth entry, or 0 if non-existent. Do not change **environ.release**. See also system variables **_RELEASE** and **environ.version**.	no
environ.version	Similar to **environ.release**, but contains version information represented by a float, not including the lib/library.agn patch level.	no
environ.withprotected	A set of names (passed as strings) that cannot by overwritten by the **with** function. Currently the names `next`, `print`, `with`, `write`, `read`, `writeline` have been assigned.	yes

System variable	Meaning	Write
environ.withverbose	If set to **false**, the **with** function will not display warnings, the initialisation string, and the short names assigned. Default is **true**.	yes

_G	A table holding all currently assigned global names and their values, and itself. You can add or delete entries by simple table assignment or unassignment, e.g. to delete the **print** function in the current session, just enter:
	
	`> delete print from _G`
	
	`> print('Klöße !')`
	`Error in stdin, at line 1:`
	`attempt to call global `print` (a null value)`
		yes

_PROMPT	Defines the prompt Agena displays at the console. If unassigned, by default the prompt is `> `.	yes
_RELEASE	Release information on the installed Agena release, returned as a string, e.g. 'AGENA >> 2.2.0'. See also system variables **environ.release** and **environ.version**.	no
nargs	Number of arguments actually passed in a function call, including varargs	no
procname	Refers to the function currently invoked, can be used in recursive calls	no

Table 22: System variables

All environ.* settings are reset by the **restart** statement to their original defaults, whereas those settings the user defines with the **environ.kernel** function will never be modified or deleted by a **restart**.

Some of the default settings can be found at the bottom of the lib/library.agn file.

See also:

- Chapter 14.2 for a description of the **kernel** functions for other settings.
- Appendix A5 for settings that control how Agena outputs data at the console.

A5 Command-Line Usage & Scripting

Agena can be used in the command line as follows:

```
agena [options] [script [arguments]]
```

This means that any option, a script name, and the arguments are all optional. If you just enter

```
shell> agena
```

Agena is started in interactive mode immediately.

There are two ways to run an Agena script with some arguments and then return to the command line immediately without entering interactive mode:

A5.1 Using the -e Option

We may write a script with a text editor, e.g. one to print the sine of a number, for example the following two lines:

```
n := n or Pi;  # if n is not set from the shell, just assign Pi to n
writeline(sin(n));
```

When using the -e option, we first assign the desired number to a variable and then call the script by its name:

```
shell> agena -e "n := Pi/2" sin.agn
1
```

Note that you first have to enter the -e option along with the assignment statement, and then the name of the script.

A much better alternative is this:

A5.2 Using the Internal args Table and Exit Status

Everything you pass to the interpreter from the command line is stored in the **args** table.

The name of the script is always stored at index 0, the arguments are stored at the positive indices 1, 2, etc., in the order given by the user. The name of the Agena binary and any options are accessible via negative keys. The name of the interpreter binary is always at the smallest index.

Consider the following script called 'args.agn':

```
for i, j in args do
   writeline(i, j, delim='\t')
od;
```

If you run it, the output will be:

```
shell> agena -m args.agn 1.1 2.2 3.3
4'194'303 KBytes of physical RAM free.


1       1.1
2       2.2
3       3.3
-2      agena
-1      -m
0       args.agn
```

Just play around with this a little bit.

Let us use our new knowledge: The script 'ln.agn' requires at least one number and calculates the corresponding natural logarithm. The numbers entered at the command line are put into the **args** table as strings, so you should convert them back into numbers first. The number of actual arguments - without script name, options and Agena binary file name - are stored in **nargs**. **os.exit** passes an exit code to the shell, if needed.

```
# Evaluate the natural logarithm of given numbers

if nargs < 1 then
   print('Error, need at least one number');
   os.exit(-1)  # just return error exit code
fi

rc := 0;  # exit/return code of the script

for i to nargs do  # iterate each argument at the command line
   x := tonumber(args[i]);
   if x :- number then x := 0 fi;  # if conversion failed, set x to 0
   r := ln(x);
   if r = undefined then rc := 1 fi;  # there was an arithmetic error
   writeline('ln(', x, ') = ', r)
od;

/* clear interpreter state, perform garbage collection & return exit code:
   -1 = no arguments given
    0 = okay
    1 = domain error or wrong type of argument */

os.exit(rc, true);
```

Use it:

```
shell> agena ln.agn 0 1 2
ln(0) = undefined
ln(1) = 0
ln(2) = 0.69314718055995
```

You will find sample scripts in the 'share/scripting' directory of your Agena distribution. The folder also includes batch files to start the scripts from a shell in OS/2 Warp 4.5 (e.g. 'whereis.cmd'), DOS and Windows (e.g. 'whereis.bat').

A5.3 Running a Script and then Entering Interactive Mode

The `-i` option allows you to enter the interactive level after running a script or passing other options to Agena. The position of the `-i` option is free. The following shell statement resets the Agena prompt and starts the interpreter:

```
shell> agena -i -e "_PROMPT := 'AGENA> '"
AGENA>
```

A5.4 Running Scripts in UNIX and Mac OS X

If you use Agena in UNIX and Mac OS X, then you can execute Agena scripts directly by just entering the name of the script followed by any arguments (if needed).

Just insert the following line at the head, i.e. the very first line, of each script:

```
#!/usr/local/bin/agena
```

and set the appropriate rights for the script file (e.g. `chmod a+x scriptname`).
An example:

```
bash> ./sin.agn 1
0.8414709848079
```

In all other operating systems, the first line is ignored by the interpreter, so you do not have to delete the first line of the script in order to use scripts you have originally written under UNIX or Mac.

Please make sure that the file is stored in UNIX line break format - and not Windows line breaks.

A5.5 Command Line Switches

The available switches are:

Option	Function
-e "stat"	execute string "stat" (double quotes needed)
-h	help information
-i	enter interactive mode after executing `script` or other options
-l	print licence information
-m	print the amount of free RAM at start-up
-n	do not run initialisation file(s) agena.ini / .agenainit at start-up or **restart**
-p path	sets <path> to **libname**, overriding the standard initialisation procedure for this environment variable. The path does not need to be put in quotes if it does not contain spaces.
-r name	readlib library <name>. The name of the library does not need to be put in quotes.
-a	ignore AGENAPATH environment variable, setting **libname** by searching the file system.
-s "text"	issue the slogan "text" at start-up
-v	show version information and compilation time
-x	does not read the main library file lib/library.agn art start-up or **restart**
--	stop handling options
-	execute stdin and stop handling options

Table 23: Command line options

Instead of a preceding hyphen you can also use a slash, e.g. `agena /d` and `agena -d` for debugging mode are accepted.

A6 Define Your Own Printing Rules for Types

You can tell Agena how to output strings, tables, sets, sequences, pairs, and complex values at the console.

With each call to the internal printing routine, the interpreter uses the respective **environ.aux.print*** function or settings defined in the lib/library.agn file. You may change these functions or settings according to your needs.

Table index	Type	Functionality
environ.aux.printtable	function	defines how to print a table, overriding the built-in default
environ.aux.printlongtable	function	defines how to print a table if **kernel/longtable** has been set true
environ.aux.printset	function	defines how to print a set, overriding the built-in default
environ.aux.printsequence	function	defines how to print a sequence, overriding the built-in default
environ.aux.printpair	function	defines how to print a pair, overriding the built-in default

Table index	Type	Functionality
environ.aux.printcomplex	function	defines how to print a complex value, overriding the built-in default

Table index	Type	Functionality
environ.aux. printenclosestrings	string	if set, Agena outputs strings with the prepending and appending string that is assigned to environ.aux.printenclosestrings
environ.aux.printprocedure	function	defines how to print a procedure, overriding the built-in default

Table 24: Printing functions

Alternative **environ.aux.print*** functions might look like the following one:

```
> environ.aux.printset := proc(s) is
>    write('set(');
>    if size s > 0 then
>       for i in s do
>          write(i, ', ');
>       od;
>       write('\b\b');
>    fi;
>    write(')');
> end;

> environ.aux.printcomplex := proc(s) is
>    write('cmplx(', real(s), ', ', imag(s), ')');
> end;

> {1, 2}:
set(1, 2)

> 1*2*I:
cmplx(1, 2)
```

A7 The Agena Initialisation File

You can customise your personal Agena environment via special initialisation files.

The initialisation files may include code written agena and will always be executed when Agena is started or **restart**ed. They can include definitions or redefinitions of predefined (environment) variables, and feature self-written procedures or statements to be executed at start-up.

Two kinds of initialisation files are supported:

1. a global initialisation file, and
2. a personal initialisation file for the current user.

Agena first tries to read the global initialisation file, and then the user's initialisation file. If the initialisation files do not exist, nothing happens and Agena starts without errors.

The global initialisation file should reside in the `lib` folder of your Agena installation and is always named `agena.ini` for all operating systems. You may find your Agena

installation in `/usr/agena` on UNIX platforms, and usually in `<drive:>/Program Files/Agena` or `<drive:>/Program Files(x86)/Agena` on Windows systems.

In Solaris, Linux, Mac OS X, the personal initialisation file resides in the folder pointed to be the `HOME` environment variable. The personal Agena initialisation file on UNIX machines is called `.agenainit` (not `agena.ini`). Thus the path is `$HOME/.agenainit`.

In Windows, the system environment variable `UserProfile` points to the user's home folder, and the personal initialisation file is called `agena.ini`, (not `.agenainit`), thus the file path is `%UserProfile%/agena.ini`.

On Windows platforms, the user's initialisation file should be put into the user's respective home folder:

Windows version	Path to user's home directory
NT 4.0	`<drive:>\WINNT\Profiles\<username>`
2000, XP, 2003	`<drive:>\Documents and Settings\<username>`
Vista and later	`<drive:>\Users\<username>`

Table 25: Windows' `home` paths

In OS/2 and DOS, Agena tries to find the user's personal `agena.ini` file in the directory pointed to by the environment variable `HOME`, if it has been defined. If HOME has not been defined, it searches in the folder pointed to by the environment variable `USER`, if the latter has been defined. Otherwise, the personal file is not read.

Agena is shipped with a file called `agena.ini.sample` that resides in the `lib` folder of your installation. You can rename it to `agena.ini` or `.agenainit` and play with it - but beware not to overwrite the initialisation which you may already have created.

Here is a sample file:

```
###########################################################################
#                                                                         #
# Agena initialisation file                                               #
#                                                                         #
###########################################################################

# assign short names for the following library functions:
execute := os.execute;

###########################################################################
# Extend libname to include paths to additional libraries (but only       #
# if directories exist)                                                    #
###########################################################################

if os.isWin() or os.isOS2() or os.isDOS() then
   addpaths := seq(
     'd:/agena/phq',
     'd:/agena/pcomp'
   )
elif os.isSolaris() then
   addpaths := seq(
     '/export/home/proglang/agena/phq',
     '/export/home/proglang/agena/pcomp'
```

```
    )
elif os.isLinux() then
   addpaths := seq(
       '~/agena/phq',
       '~/agena/pcomp'
   )
fi;

for i in addpaths do
   if os.exists(i) and i in libname = null then
      libname := libname & ';' & i
   fi
od;

clear addpaths;

writeline('Have fun with Agena !\n');


########################################################################
# Set default plotting options for gdi.plotfn                         #
########################################################################

import gdi;
gdi.setoptions(colour~'red', axescolour~'grey');
```

A8 Escape Sequences

Agena supports the following escape sequences known from ANSI C:

Sequence	Meaning
\a	alert
\b	backspace
\f	formfeed
\n	new line
\r	carriage return
\t	horizontal tabulator
\v	vertical tabulator
\xAB	hex escape
\z	skips subsequent white-space characters, including line breaks; it is particularly useful to break and indent a long literal string into multiple lines without adding the newlines and spaces into the string contents

Table 26: Escape sequences

```
> print('{\n\z
>       \x20   \"\104ello\": \"world\"\n\z
>       }')
{
    "hello": "world"
}
```

A9 Backward Compatibility

Aliases for deprecated functions in Agena versions prior to 1.0 are no longer automatically initialised at start-up. However, by entering

```
> import compat;
```

you can activate them in your current session if you prefer compatibility to Agena 1.0. For all other cases, please consult the `change.log` file distributed with the source and binary editions.

This concerns all deprecated function names in the base library, in the **math**, **package**, **strings**, **tables**, **utils** packages, as well as the former **_Env*** environment control variables.

Deprecated names of functions in the **linalg** package can only be used by uncommenting the alias assignments at the bottom of the lib/linalg.agn file.

Users of the **mapm** package should first **import** the **mapm** package and then load the compat.agn file.

A10 Some Few Technical Notes

All Solaris and RedHat binaries of Agena have been created with GCC 4.4.5, the Debian x86 binaries with GCC 6.3.0 and the AMD64 binaries with GCC 11.2.

All OS/2 binaries have been created with Paul Smith's GCC 4.4.6.

All Windows binaries of Agena have been created with MinGW/GCC 6.3.0.

All Mac OS X binaries of Agena have been created with Apple's GCC 4.2.1.

The DOS version is being compiled with DJGPP/GCC 12.1.0.

The C Sources should be ANSI C99 compatible, mostly due to Agena's support of complex arithmetic.

Appendix B

B1 Agena Licence

The Agena source code is licenced under the terms of the following licence:

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicence, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notices and this permission notice shall be included in all copies or portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS' WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.

IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

B2 GNU GPL v2 Licence

The Solaris, Linux, Windows, OS/2, Mac OS X, and DOS binaries are distributed under the GNU GPL v2 licence reproduced below:

GNU GENERAL PUBLIC LICENSE
Version 2, June 1991

Copyright (C) 1989, 1991 Free Software Foundation, Inc.,
51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA

Everyone is permitted to copy and distribute verbatim copies of this licence document, but changing it is not allowed.

Preamble

   The licences for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public Licence is intended to guarantee your freedom to share and change free software--to make sure the software is free for all its users. This General Public Licence applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by

the GNU Lesser General Public Licence instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licences are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this licence which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licences, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licenced for everyone's free use or not licenced at all.

The precise terms and conditions for copying, distribution and modification follow.

<div align="center">

GNU GENERAL PUBLIC LICENSE
TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

</div>

0. This Licence applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public Licence. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licencee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this Licence; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this Licence and to the absence of any warranty; and give any other recipients of the Program a copy of this Licence along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

a. You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.
b. You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licenced as a whole at no charge to all third parties under the terms of this Licence.
c. If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this Licence. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this Licence, and its terms, do not apply to those sections when you distribute them as separate works. But when

you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this Licence, whose permissions for other licencees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this Licence.

   3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

a. Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
b. Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
c. Accompany it with the information you received as to the offer to distribute corresponding source code.   (This alternative is allowed only for non-commercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

   4. You may not copy, modify, sublicence, or distribute the Program except as expressly provided under this Licence. Any attempt otherwise to copy, modify, sublicence or distribute the Program is void, and will automatically terminate your

rights under this Licence. However, parties who have received copies, or rights, from you under this Licence will not have their licences terminated so long as such parties remain in full compliance.

   5. You are not required to accept this Licence, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this Licence. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this Licence to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

   6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a licence from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this Licence.

   7. If, as a consequence of a court judgement or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this Licence, they do not excuse you from the conditions of this Licence. If you cannot distribute so as to satisfy simultaneously your obligations under this Licence and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent licence would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this Licence would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public licence practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licencee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this Licence.

   8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places

the Program under this Licence may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this Licence incorporates the limitation as if written in the body of this Licence.

  9. The Free Software Foundation may publish revised and/or new versions of the General Public Licence from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this Licence which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this Licence, you may choose any version ever published by the Free Software Foundation.

  10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

  11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

  12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS
How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

    <one line to give the program's name and a brief idea of what it does.>
    Copyright (C) <year> <name of author>

    This program is free software; you can redistribute it and/or modify
    it under the terms of the GNU General Public Licence as published by
    the Free Software Foundation; either version 2 of the Licence, or
    (at your option) any later version.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
    GNU General Public Licence for more details.

    You should have received a copy of the GNU General Public Licence along
    with this program; if not, write to the Free Software Foundation, Inc.,
    51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA.

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

    Gnomovision version 69, Copyright (C) year name of author
    Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type `show w'.
    This is free software, and you are welcome to redistribute it
    under certain conditions; type `show c' for details.

The hypothetical commands `show w' and `show c' should show the appropriate parts of the General Public Licence. Of course, the commands you use may be called something other than `show w' and `show c'; they could even be mouse-clicks or menu items--whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

Yoyodyne, Inc., hereby disclaims all copyright interest in the program
`Gnomovision' (which makes passes at compilers) written by James Hacker.

<signature of Ty Coon>, 1 April 1989
Ty Coon, President of Vice

This General Public Licence does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public Licence instead of this Licence.

B3 Sun Microsystems Licence for the fdlibm IEEE 754 Style Arithmetic Library

```
 * ====================================================
 * Copyright (C) 1993 by Sun Microsystems, Inc. All rights reserved.
 *
 * Developed at SunPro, a Sun Microsystems, Inc. business.
 * Permission to use, copy, modify, and distribute this
 * software is freely granted, provided that this notice
 * is preserved.
 * ====================================================
```

B4 GNU Lesser General Public Licence

Agena uses the g2 graphic library which is distributed under the GNU LGPL v2.1 licence reproduced below:

GNU LESSER GENERAL PUBLIC LICENSE
Version 2.1, February 1999

Copyright (C) 1991, 1999 Free Software Foundation, Inc.
  59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
Everyone is permitted to copy and distribute verbatim copies
of this licence document, but changing it is not allowed.

[This is the first released version of the Lesser GPL. It also counts as the successor of the GNU Library Public Licence, version 2, hence the version number 2.1.]

Preamble

The licences for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public Licences are intended to guarantee your freedom to share and change free software--to make sure the software is free for all its users.

This licence, the Lesser General Public Licence, applies to some specially designated software packages--typically libraries--of the Free Software Foundation and other authors who decide to use it. You can use it too, but we suggest you first think carefully about whether this licence or the ordinary General Public Licence is the better strategy to use in any particular case, based on the explanations below.

When we speak of free software, we are referring to freedom of use, not price. Our General Public Licences are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish); that you receive source code or can get it if you want it; that you can change the software and use pieces of it in new free programs; and that you are informed that you can do these things.

To protect your rights, we need to make restrictions that forbid distributors to deny you these rights or to ask you to surrender these rights. These restrictions translate to certain responsibilities for you if you distribute copies of the library or if you modify it.

For example, if you distribute copies of the library, whether gratis or for a fee, you must give the recipients all the rights that we gave you. You must make sure that they, too, receive or can get the source code. If you link other code with the library, you must provide complete object files to the recipients, so that they can relink them with the library after making changes to the library and recompiling it. And you must show them these terms so they know their rights.

We protect your rights with a two-step method: (1) we copyright the library, and (2) we offer you this licence, which gives you legal permission to copy, distribute and/or modify the library.

To protect each distributor, we want to make it very clear that there is no warranty for the free library. Also, if the library is modified by someone else and passed on, the recipients should know that what they have is not the original version, so that the original author's reputation will not be affected by problems that might be introduced by others.

Finally, software patents pose a constant threat to the existence of any free program. We wish to make sure that a company cannot effectively restrict the users of a free program by obtaining a restrictive licence from a patent holder. Therefore, we insist that any patent licence obtained for a version of the library must be consistent with the full freedom of use specified in this licence.

Most GNU software, including some libraries, is covered by the ordinary GNU General Public Licence. This licence, the GNU Lesser General Public Licence, applies to certain designated libraries, and is quite different from the ordinary General Public Licence. We use this licence for certain libraries in order to permit linking those libraries into non-free programs.

When a program is linked with a library, whether statically or using a shared library, the combination of the two is legally speaking a combined work, a derivative of the original library. The ordinary General Public Licence therefore permits such linking only if the entire combination fits its criteria of freedom. The Lesser General Public Licence permits more lax criteria for linking other code with the library.

We call this licence the "Lesser" General Public Licence because it does Less to protect the user's freedom than the ordinary General Public Licence. It also

provides other free software developers Less of an advantage over competing non-free programs. These disadvantages are the reason we use the ordinary General Public Licence for many libraries. However, the Lesser license provides advantages in certain special circumstances.

For example, on rare occasions, there may be a special need to encourage the widest possible use of a certain library, so that it becomes a de-facto standard. To achieve this, non-free programs must be allowed to use the library. A more frequent case is that a free library does the same job as widely used non-free libraries. In this case, there is little to gain by limiting the free library to free software only, so we use the Lesser General Public Licence.

In other cases, permission to use a particular library in non-free programs enables a greater number of people to use a large body of free software. For example, permission to use the GNU C Library in non-free programs enables many more people to use the whole GNU operating system, as well as its variant, the GNU/Linux operating system.

Although the Lesser General Public Licence is Less protective of the users' freedom, it does ensure that the user of a program that is linked with the Library has the freedom and the wherewithal to run that program using a modified version of the Library.

The precise terms and conditions for copying, distribution and modification follow. Pay close attention to the difference between a "work based on the library" and a "work that uses the library". The former contains code derived from the library, whereas the latter must be combined with the library in order to run.

GNU LESSER GENERAL PUBLIC LICENSE
TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This Licence Agreement applies to any software library or other program which contains a notice placed by the copyright holder or other authorized party saying it may be distributed under the terms of this Lesser General Public Licence (also called "this Licence"). Each licensee is addressed as "you".

A "library" means a collection of software functions and/or data prepared so as to be conveniently linked with application programs (which use some of those functions and data) to form executables.

The "Library", below, refers to any such software library or work which has been distributed under these terms. A "work based on the Library" means either the Library or any derivative work under copyright law: that is to say, a work containing the Library or a portion of it, either verbatim or with modifications and/or translated straightforwardly into another language. (Hereinafter, translation is included without limitation in the term "modification".)
"Source code" for a work means the preferred form of the work for making modifications to it. For a library, complete source code means all the source code

for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the library.

Activities other than copying, distribution and modification are not covered by this Licence; they are outside its scope. The act of running a program using the Library is not restricted, and output from such a program is covered only if its contents constitute a work based on the Library (independent of the use of the Library in a tool for writing it). Whether that is true depends on what the Library does and what the program that uses the Library does.

1. You may copy and distribute verbatim copies of the Library's complete source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this Licence and to the absence of any warranty; and distribute a copy of this Licence along with the Library.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Library or any portion of it, thus forming a work based on the Library, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

a. The modified work must itself be a software library.
b. You must cause the files modified to carry prominent notices stating that you changed the files and the date of any change.
c. You must cause the whole of the work to be licensed at no charge to all third parties under the terms of this Licence.
d. If a facility in the modified Library refers to a function or a table of data to be supplied by an application program that uses the facility, other than as an argument passed when the facility is invoked, then you must make a good faith effort to ensure that, in the event an application does not supply such function or table, the facility still operates, and performs whatever part of its purpose remains meaningful.

   (For example, a function in a library to compute square roots has a purpose that is entirely well-defined independent of the application. Therefore, Subsection 2d requires that any application-supplied function or table used by this function must be optional: if the application does not supply it, the square root function must still compute square roots.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Library, and can be reasonably considered independent and separate works in themselves, then this Licence, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Library, the distribution of the whole must be on the terms of this Licence, whose

permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Library.

In addition, mere aggregation of another work not based on the Library with the Library (or with a work based on the Library) on a volume of a storage or distribution medium does not bring the other work under the scope of this Licence.

  3. You may opt to apply the terms of the ordinary GNU General Public Licence instead of this Licence to a given copy of the Library. To do this, you must alter all the notices that refer to this Licence, so that they refer to the ordinary GNU General Public Licence, version 2, instead of to this Licence. (If a newer version than version 2 of the ordinary GNU General Public Licence has appeared, then you can specify that version instead if you wish.) Do not make any other change in these notices.
  Once this change is made in a given copy, it is irreversible for that copy, so the ordinary GNU General Public Licence applies to all subsequent copies and derivative works made from that copy.

  This option is useful when you wish to copy part of the code of the Library into a program that is not a library.

  4. You may copy and distribute the Library (or a portion or derivative of it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange.

  If distribution of object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place satisfies the requirement to distribute the source code, even though third parties are not compelled to copy the source along with the object code.

  5. A program that contains no derivative of any portion of the Library, but is designed to work with the Library by being compiled or linked with it, is called a "work that uses the Library". Such a work, in isolation, is not a derivative work of the Library, and therefore falls outside the scope of this Licence.

  However, linking a "work that uses the Library" with the Library creates an executable that is a derivative of the Library (because it contains portions of the Library), rather than a "work that uses the library". The executable is therefore covered by this Licence. Section 6 states terms for distribution of such executables.
  When a "work that uses the Library" uses material from a header file that is part of the Library, the object code for the work may be a derivative work of the Library even though the source code is not. Whether this is true is especially significant if the

work can be linked without the Library, or if the work is itself a library. The threshold for this to be true is not precisely defined by law.

If such an object file uses only numerical parameters, data structure layouts and accessors, and small macros and small inline functions (ten lines or less in length), then the use of the object file is unrestricted, regardless of whether it is legally a derivative work. (Executables containing this object code plus portions of the Library will still fall under Section 6.)

Otherwise, if the work is a derivative of the Library, you may distribute the object code for the work under the terms of Section 6. Any executables containing that work also fall under Section 6, whether or not they are linked directly with the Library itself.

6. As an exception to the Sections above, you may also combine or link a "work that uses the Library" with the Library to produce a work containing portions of the Library, and distribute that work under terms of your choice, provided that the terms permit modification of the work for the customer's own use and reverse engineering for debugging such modifications.

You must give prominent notice with each copy of the work that the Library is used in it and that the Library and its use are covered by this Licence. You must supply a copy of this Licence. If the work during execution displays copyright notices, you must include the copyright notice for the Library among them, as well as a reference directing the user to the copy of this Licence. Also, you must do one of these things:

a. Accompany the work with the complete corresponding machine-readable source code for the Library including whatever changes were used in the work (which must be distributed under Sections 1 and 2 above); and, if the work is an executable linked with the Library, with the complete machine-readable "work that uses the Library", as object code and/or source code, so that the user can modify the Library and then relink to produce a modified executable containing the modified Library. (It is understood that the user who changes the contents of definitions files in the Library will not necessarily be able to recompile the application to use the modified definitions.)
b. Use a suitable shared library mechanism for linking with the Library. A suitable mechanism is one that (1) uses at run time a copy of the library already present on the user's computer system, rather than copying library functions into the executable, and (2) will operate properly with a modified version of the library, if the user installs one, as long as the modified version is interface-compatible with the version that the work was made with.
c. Accompany the work with a written offer, valid for at least three years, to give the same user the materials specified in Subsection 6a, above, for a charge no more than the cost of performing this distribution.
d. If distribution of the work is made by offering access to copy from a designated place, offer equivalent access to copy the above specified materials from the same place.

e. Verify that the user has already received a copy of these materials or that you have already sent this user a copy.

   For an executable, the required form of the "work that uses the Library" must include any data and utility programs needed for reproducing the executable from it. However, as a special exception, the materials to be distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

   It may happen that this requirement contradicts the license restrictions of other proprietary libraries that do not normally accompany the operating system. Such a contradiction means you cannot use both them and the Library together in an executable that you distribute.

   7. You may place library facilities that are a work based on the Library side-by-side in a single library together with other library facilities not covered by this Licence, and distribute such a combined library, provided that the separate distribution of the work based on the Library and of the other library facilities is otherwise permitted, and provided that you do these two things:

a. Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities. This must be distributed under the terms of the Sections above.
b. Give prominent notice with the combined library of the fact that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

   8. You may not copy, modify, sublicense, link with, or distribute the Library except as expressly provided under this Licence. Any attempt otherwise to copy, modify, sublicense, link with, or distribute the Library is void, and will automatically terminate your rights under this Licence. However, parties who have received copies, or rights, from you under this Licence will not have their licenses terminated so long as such parties remain in full compliance.

   9. You are not required to accept this Licence, since you have not signed it. However, nothing else grants you permission to modify or distribute the Library or its derivative works. These actions are prohibited by law if you do not accept this Licence. Therefore, by modifying or distributing the Library (or any work based on the
Library), you indicate your acceptance of this Licence to do so, and all its terms and conditions for copying, distributing or modifying the Library or works based on it.

   10. Each time you redistribute the Library (or any work based on the Library), the recipient automatically receives a license from the original licensor to copy, distribute, link with or modify the Library subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights

granted herein. You are not responsible for enforcing compliance by third parties with this Licence.

   11. If, as a consequence of a court judgement or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this Licence, they do not excuse you from the conditions of this Licence. If you cannot distribute so as to satisfy simultaneously your obligations under this Licence and any other pertinent obligations, then as a consequence you may not distribute the Library at all. For example, if a patent license would not permit royalty-free redistribution of the Library by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this Licence would be to refrain entirely from distribution of the Library.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply, and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this Licence.

   12. If the distribution and/or use of the Library is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Library under this Licence may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this Licence incorporates the limitation as if written in the body of this Licence.

   13. The Free Software Foundation may publish revised and/or new versions of the Lesser General Public Licence from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Library specifies a version number of this Licence which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Library does not specify a license version number, you may choose any version ever published by the Free Software Foundation.

14. If you wish to incorporate parts of the Library into other free programs whose distribution conditions are incompatible with these, write to the author to ask for permission.   For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

15. BECAUSE THE LIBRARY IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE LIBRARY, TO THE EXTENT PERMITTED BY APPLICABLE LAW.
EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE LIBRARY "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE LIBRARY IS WITH YOU. SHOULD THE LIBRARY PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE LIBRARY AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE LIBRARY (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE LIBRARY TO OPERATE WITH ANY OTHER SOFTWARE), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Libraries

If you develop a new library, and you want it to be of the greatest possible use to the public, we recommend making it free software that everyone can redistribute and change.   You can do so by permitting redistribution under these terms (or, alternatively, under the terms of the ordinary General Public Licence).

To apply these terms, attach the following notices to the library.   It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

    <one line to give the library's name and a brief idea of what it does.>
    Copyright (C) <year> <name of author>

    This library is free software; you can redistribute it and/or

B5 SOFA Software Licence

3. You (the user) may copy and distribute SOFA source code to others, and use and adapt its code and algorithms in your own software, on a world-wide, royalty-free basis.   That portion of your distribution that does not consist of intact and unchanged copies of SOFA source code files is a "derived work" that must comply with the following requirements:

a) Your work shall be marked or carry a statement that it (i) uses routines and computations derived by you from software provided by SOFA under license to you; and (ii) does not itself constitute software provided by and/or endorsed by SOFA.

b) The source code of your derived work must contain descriptions of how the derived work is based upon, contains and/or differs from the original SOFA software.

c) The names of all routines in your derived work shall not include the prefix "iau" or "sofa" or trivial modifications thereof such as changes of case.

d) The origin of the SOFA components of your derived work must not be misrepresented; you must not claim that you wrote the original software, nor file a patent application for SOFA software or algorithms embedded in the SOFA software.

e) These requirements must be reproduced intact in any source distribution and shall apply to anyone to whom you have granted a further right to modify the source code of your derived work.

Note that, as originally distributed, the SOFA software is intended to be a definitive implementation of the IAU standards, and consequently third-party modifications are discouraged. All variations, no matter how minor, must be explicitly marked as such, as explained above.

4. You shall not cause the SOFA software to be brought into disrepute, either by misuse, or use for inappropriate tasks, or by inappropriate modification.

5. The SOFA software is provided "as is" and SOFA makes no warranty as to its use or performance. SOFA does not and cannot warrant the performance or results which the user may obtain by using the SOFA software.   SOFA makes no warranties, express or implied, as to non-infringement of third party rights, merchantability, or fitness for any particular purpose. In no event will SOFA be liable to the user for any consequential, incidental, or special damages, including any lost profits or lost savings, even if a SOFA representative has been advised of such damages, or for any claim by any third party.

6. The provision of any version of the SOFA software under the terms and conditions specified herein does not imply that future versions will also be made available under the same terms and conditions.

In any published work or commercial product which uses the SOFA software directly, acknowledgement (see www.iausofa.org) is appreciated.

Correspondence concerning SOFA software should be addressed as follows:

    By email: sofa@ukho.gov.uk
    By post: IAU SOFA Center
               HM Nautical Almanac Office
               UK Hydrographic Office
               Admiralty Way, Taunton
               Somerset, TA1 2DN
               United Kingdom

B6 MAPM Copyright Remark (Mike's Arbitrary Precision Math Library)

Copyright (C) 1999 - 2007   Michael C. Ring

This software is Freeware.

Permission to use, copy, and distribute this software and its documentation for any purpose with or without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation.

Permission to modify the software is granted. Permission to distribute the modified code is granted. Modifications are to be distributed by using the file 'license.txt' as a template to modify the file header. 'license.txt' is available in the official MAPM distribution.

To distribute modified source code, insert the file 'license.txt' at the top of all modified source code files and edit accordingly.

This software is provided "as is" without express or implied warranty.

B7 RSA Security/MD5 Licence

Copyright (C) 1990, RSA Data Security, Inc. All rights reserved.

License to copy and use this software is granted provided that it is identified as the "RSA Data Security, Inc. MD5 Message Digest Algorithm" in all material mentioning or referencing this software or this function.

License is also granted to make and use derivative works provided that such works are identified as "derived from the RSA Data Security, Inc. MD5 Message Digest Algorithm" in all material mentioning or referencing the derived work.

RSA Data Security, Inc. makes no representations concerning either the merchantability of this software or the suitability of this software for any particular purpose. It is provided "as is" without express or implied warranty of any kind.

These notices must be retained in any copies of any part of this documentation and/or software.

B8 David Schultz's Openlibm Licence

Copyright (c) 2011 David Schultz <das@FreeBSD.ORG>
All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:
1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

B9 ISC Licence

Copyright (c) 2005-2008, Simon Howard

Permission to use, copy, modify, and/or distribute this software for any purpose with or without fee is hereby granted, provided that the above copyright notice and this permission notice appear in all copies.

THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY SPECIAL, DIRECT, INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

B10 Other Copyright Remarks

The Solaris, Linux, Mac OS X, and Windows binaries include code from the gd package which has been published with the following copyright notices:

Portions copyright 1994, 1995, 1996, 1997, 1998, 1999, 2000, 2001, 2002 by Cold Spring Harbor Laboratory. Funded under Grant P41-RR02188 by the National Institutes of Health.

Portions copyright 1996, 1997, 1998, 1999, 2000, 2001, 2002 by Boutell.Com, Inc.

Portions relating to GD2 format copyright 1999, 2000, 2001, 2002 Philip Warner.

Portions relating to PNG copyright 1999, 2000, 2001, 2002 Greg Roelofs.

Portions relating to gdttf.c copyright 1999, 2000, 2001, 2002 John Ellson (ellson@lucent.com).

Portions relating to gdft.c copyright 2001, 2002 John Ellson (ellson@lucent.com).

Portions copyright 2000, 2001, 2002, 2003, 2004, 2005, 2006, 2007 Pierre-Alain Joye (pierre@libgd.org).

Portions relating to JPEG and to color quantization copyright 2000, 2001, 2002, Doug Becker and copyright (C) 1994, 1995, 1996, 1997, 1998, 1999, 2000, 2001, 2002, Thomas G. Lane. This software is based in part on the work of the Independent JPEG Group. See the file README-JPEG.TXT for more information.

Portions relating to WBMP copyright 2000, 2001, 2002 Maurice Szmurlo and Johan Van den Brande.

Permission has been granted to copy, distribute and modify gd in any context without fee, including a commercial application, provided that this notice is present in user-accessible supporting documentation.

This does not affect your ownership of the derived work itself, and the intent is to assure proper credit for the authors of gd, not to interfere with your productive use of gd. If you have questions, ask. "Derived works" includes all programs that utilise the library. Credit must be given in user-accessible documentation.

This software is provided "AS IS." The copyright holders disclaim all warranties, either express or implied, including but not limited to implied warranties of merchantability and fitness for a particular purpose, with respect to this code and accompanying documentation.

Appendix C

C1: Further Reading

A selection of books that helped a lot in recent years when developing Agena:

- Niklaus Wirth: Algorithmen und Datenstrukturen mit Modula-2
- Roberto Ierusalimschy: Programming in Lua
- Roberto Ierusalimschy, Luiz Henrique de Figueirido, Waldemar Celes: Lua 5.1 Reference Manual
- Kurt Jung & Aaron Brown: Beginning Lua Programming
- Jürgen Wolf: C von A bis Z
- Brian W. Kernighan & Dennis M. Ritchie: The C Programming Language
- Federico Biancuzzi & Shane Warden (Ed.): Masterminds of Programming
- Michael. B. Monagan, Keith O. Geddes, K. M. Heal, G. Labahn, S. M. Vorkoetter, J. McCarron, P. DeMarco: Maple 7 Programming Guide
- Brian "Beej Jorgensen" Hall, Beej's Guide to Network Programming, Using Internet Sockets
- Jan Jones: QL SuperBASIC - The Definitive Handbook
- Frank G. Pagan: A Practical Guide to Algol68

Index

G