agena >>

a programming language

primer and reference
for version 1.12.3

by alexander walz

july 21, 2013

The latest release of Agena can be found at http://sourceforge.net/projects/agena.

This manual has been created with Lotus Word Pro 98, yWorks yEd Graph Editor 3.9.2, and PDF Creator 1.2.3.

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

Many portions of Chapter 7 have 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.

math.nextafter

uses a modified version of the C function nextafter that has originally been published by Sun Microsystems with the fdlibm IEEE 754 floating-point C library. The author of the modifications is unknown, but the modified code can be found at http://www.koders.com (file s_nextafter.c). 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**, **calc.fsum**, **nseq**, **stats.amean**, **skycrane.count**, **stats.cumsum**, and **stats.sum**.

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 mostly uses a modified version of the Kahan algorithm developed by Kazufumi Ozawa, published in his paper `Analysis and Improvement of Kahan's 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.

besselj, bessely

The complex versions of the functions use procedures 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 binding to the g2 graphical library 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 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. The function has been originally written for and currently maintained as a part of popa3d, a POP3 server: http://www.openwall.com/popa3d/.

Functions **arctan**, **expx2**, **gamma**, **lngamma**, **calc.dawson**, **calc.dilog**, **calc.Ci**, **calc.Chi**, **calc.Ei**, **calc.fresnelc**, **calc.fresnels**, **calc.Psi**, **calc.Si**, **calc.Shi**, and **calc.Ssi**

use algorithms written in ANSI C by Stephen L. Moshier for the Cephes Math Library Release 2.9 as of June, 2000. Copyright by Stephen L. Moshier.

erf, erfc, 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 library. The Shapelib library has been published under the MIT licence.

AgenaEdit GUI

The GUI is based on an editor published under the GPL licence and written by Bill Spitzak and others for FLTK 1.3 http://www.fltk.org. Thanks to Albrecht Schlosser for making the editor work with Agena.

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 Lorenzo Seidenari 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.

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.

stats.deltalist, stats.cumsum, stats.colnorm, stats.rownorm, stats.sum

These functions have been inspired by the deltaList, and cumulativeSum, colNorm, and rowNorm 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

taken from an Excel file import library written for Modula-2. The original Modula-2 procedure had presumably been published at modula2.org in 2004. There have not been any copyright remarks in ExcelIn.zip.

astro.jdate

uses a C routine 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, **polar**, **stats.cdf**, **stats.numbcomb**, **stats.numbperm**, and **stats.pdf**

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

E

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

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

Many thanks to the Lua team at PUC-Rio, Brasil, and to Agena users in Israel, Italy, Australia, Palestine, Canada, and to many other users of various nations.

Table of Contents

Chapter One

Introduction

1 Introduction

1.1 Abstract

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

Agena provides fast 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 & eComStation, Mac OS X, Haiku, and DOS.

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

http://sourceforge.net/projects/agena.

1.2 Features

Agena combines features of Lua 5, Maple, Algol 60, Algol 68, ABC, SQL, ANSI C, Sinclair ZX Spectrum BASIC, and SuperBASIC for Sinclair QL.

Agena supports all of the common functionality found in imperative languages:

- assignments,
- loops,
- conditions,
- procedures.

Besides providing these basic operations, it 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,
- an arbitrary precision mathematical library,
- 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, pairs, and complex numbers. With all of these types, you can build fast 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,
- **is** operator to return alternative values,
- numerical **for/from/to/by** loops with optional start and step values, and automatic round-off error correction of iteration variables,
- combined **for/while** loops,
- **for/in** loops over strings and complex data structures,
- **while** and **do/as** loops similar to Modula's while and repeat/until not() 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,
- 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 meaning 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, i.e. vectors, to internally store items in strict sequential order,
- pairs to hold two values or pass arguments in any order to procedures,
- threads, userdata, and lightuserdata inherited from Lua.

For performance, most basic operations on these types were 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 (the same feature is available to the above mentioned tables, sets, sequences, and pairs),
- a facility to return predefined results,
- remember tables for conducting recursion at high speed and at low memory consumption,
- closures, a features to 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, and pairs, inherited from Lua.

Some other features are:

- graphical capabilities in the Solaris, Mac, Linux, and Windows editions, provided by the **gdi** package,
- networking with the Internet and LANs,
- functions to support fast text processing (see **in**, **atendof**, **replace**, **lower**, and **upper** operators, as well as the functions in the **strings** and **utils** packages),
- easy configuration of your personal 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 (**with** function),
- 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 (**save** and **read** functions),
- 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 with the **insert**/**into** and **pop**/**from** statements,
- bitwise operators,
- direct access to the file system,
- an arbitrary precision mathematical library,
- XML import,
- xBase file support,
- a simple editor called AgenaEdit for Solaris, Linux, and Windows.

Agena is shipped with the packages mentioned above and all Lua C 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 and thus have been removed from the Lua packages. The Lua mathematical and string handling packages have been tuned and extended with new functions.

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 8.

1.4 History

I have been dreaming of creating my own programming language for the last 25 years, my first rather unsuccessful attempt made on a Sinclair ZX Spectrum in the early 1980s.

Plans became more serious 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 began with extensive modifications and extensions of the lexer, parser and the Lua Virtual Machine in summer 2007. Most of Agena's 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 yet 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 on this inconsistency issue.

Agena has been designed on Windows 2000, NT 4.0, Vista, and Windows 7 using the MinGW GCC 3.4.6 and 4.4.0 compilers. Further programming has been done on a Sun Sparc Ultra 5, a Sun Blade 150, and a Sun Blade 1500 running Solaris 10, and on openSUSE 10.3 for x86 and on Xubuntu 10.04 for Mac Mini PowerPC to make the interpreter work in UNIX environments. The original x86 Mac Version has been developed on a x86 Mac Mini. A lot of testing has been done on an Acer Aspire ONE netbook running Linpus Linux/Fedora 8.

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, surely all mathematical functions and complex arithmetic, and much, much more.

The Maple V Release 3 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. Very kind thanks to WMI's support back in the 1990s.

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,

- **case/of/esac**.

has 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.

C actually is a descendent of Algol 68.

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 the symbiosis of BASIC and Algol 60, expressed with PL/I and REXX.

Eiffel

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

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*, *jpeg*, *libgcc*, *libgd*, *libiconv*, *libintl*, *libncurses*, *libpng*, *readline*, *xpm*, and *zlib* libraries installed. From the command line, type **agena** and press RETURN.

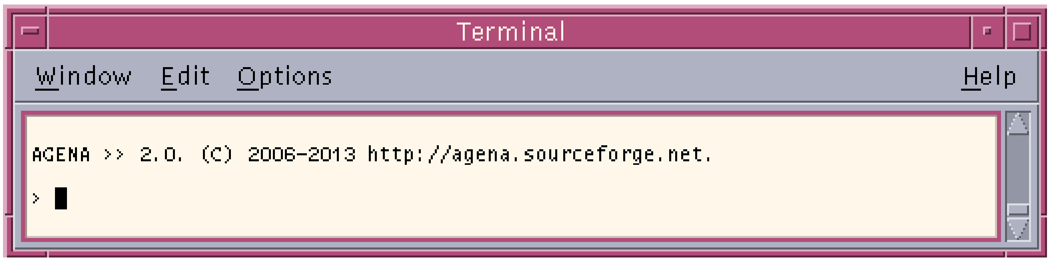


Image 1: Start-up message in Solaris

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

2.2 Linux

On Debian based distributions, install the deb installer by typing:

```
> sudo dpkg -i --force-depends agena-x.y.z-linux-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*, *jpeg*, *libgcc*, *libgd*, *libiconv*, *libintl*, *libncurses*, *libpng*, *readline*, *xpm*, and *zlib* libraries installed before.

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

The name of the Linux package is `agena`.

2.3 Windows

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

Make sure you either let the installer automatically set the environment variable called AGENAPATH containing the path to the main Agena library folder (the default) or set it later manually in the Windows Control Panel, via the `System` icon.

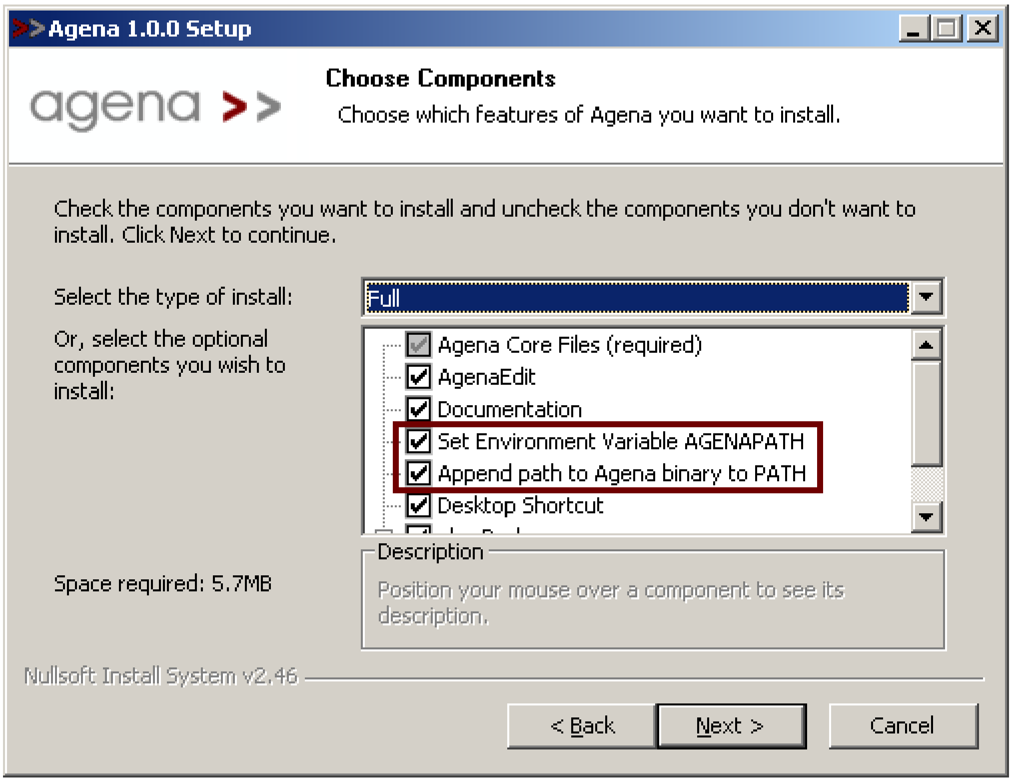


Image 2: Leave the framed settings checked

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

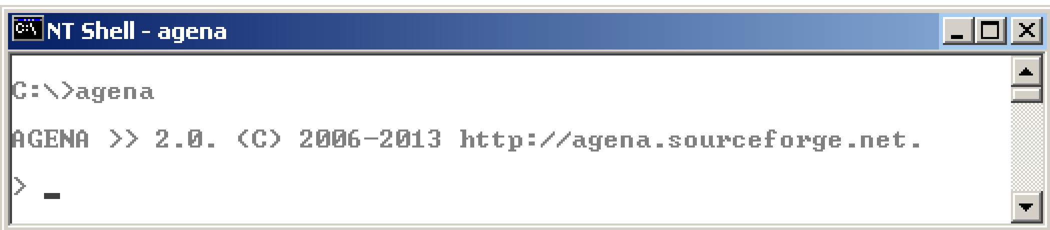


Image 3: Start-up message in Windows

Alternatively you may start **AgenaEdit**, the Agena editor and runtime environment, via the Start Menu or by typing `agenaedit` in a shell.

If you do not have admin 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.

2.4 OS/2 Warp 4 and eComStation

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

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 an OS/2 shell to run the interpreter. Agena requires EMX runtime 0.9d fix 4 or higher.

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, 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 higher

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 Haiku

Put the `agena-x.y.z-haiku.zip` file into the /boot directory and unpack it.

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

Note that you must have the *ncurses* and *readline* libraries installed before.

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

2.8 Agena Initialisation

When you start Agena, the following actions are taken:

1. The package tables for the C libraries shipped with the standard edition of Agena (e.g. math, strings, etc.) are created so that these package procedures become available to the user.

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`, 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, Haiku 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, `/boot/common/share/agena/lib` in Haiku, 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 **with** and **readlib** functions that initialize 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 the Agena language 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 - called `agena.ini` in DOS based systems and `.agenainit` in UNIX based systems including Haiku is searched by traversing all paths in **libname** from left to right. As with `library.agn`, this file contains code written in the Agena language that an administrator may customise with pre-set variables, auxiliary procedures, etc. that shall always be available to every Agena user. If the initialisation file does not exist, no error is issued. If there are multiple Agena initialisation files in your **libname** path, only the first one found is processed.

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

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

2.9 Installing Library Updates

Sometimes, library updates will be 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.read.me 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 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 become 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, Haiku, 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.

Alternatively, in Solaris, Linux, and Windows, you may start **AgenaEdit**, the Agena editor and runtime environment, by typing `agenaedit` in a shell or via the Programme Manager (Windows only).

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.

3.2 Input Conventions in AgenaEdit

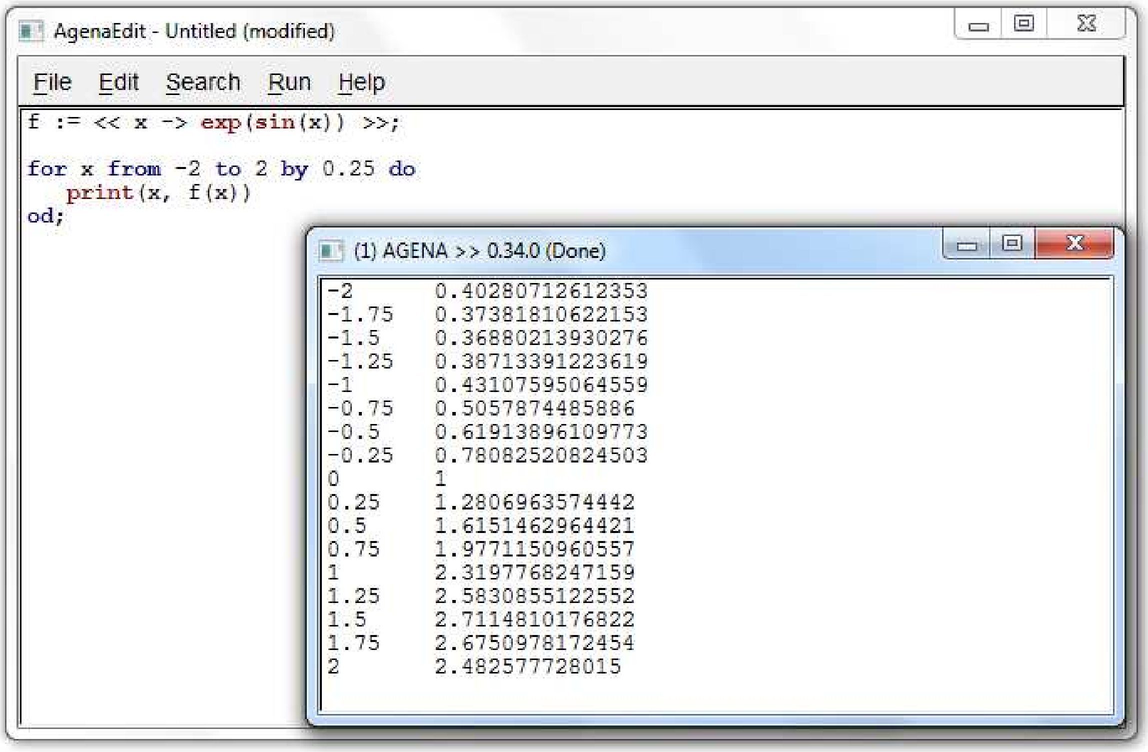
The Intel Solaris, Linux, and Windows distributions contain an editor providing syntax-highlighting and the facility to run the code you edited.

Any valid Agena code can be entered in the editor with or without a trailing semicolon.

The output of an Agena programme typed into the editor is displayed in a second window:

- Hit the F5 key to compute all statements you entered.
- Consecutive statements can be executed by selecting them and hitting the F6 key.
- 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

From this point on, this manual will deal with the console (and not AgenaEdit) edition only.

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 is 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 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 in the Solaris, Windows, UNIX, Mac OS X, Haiku, OS/2, and DOS versions by just entering the keyword **cls**[1]:

```
> cls
```

The **restart** statement[2] resets Agena to its initial state, i.e. clears all variables you defined in a session.

```
> restart
```

The **bye** statement quits a session - but could also press CTRL+C.

```
> bye
```

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

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

You may put this statement into the initialisation file in the Agena `lib` 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;
```

---

[1] The statement is not supported by AgenaEdit.

[2] dito.

Let us have a closer look at the functionality and data types available in Agena:

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 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 input 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 between single or double quotes:

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

Substrings are extracted by passing their indexes:

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

Concatenation, search, and replace operations:

```
> 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** 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**, and **xor** combine Boolean values.

```
> 1 < 2:
true

> true or false:
true
```

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 using the indexing method:

```
> 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, which indices 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 s, 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.

```
> c := 'agena';

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

3.14 Loops

A **for** loop iterates over one or more statements. It begins 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;

> print(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;

> print(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 causes another iteration of the loop to begin at once, thus skipping all of the following loop statements after the **skip** keyword for the current iteration.

The **break** statement quits the execution of the loop entirely 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
```

3.15 Procedures

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

Local variables are accessible to its 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;

> print(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:

```
> print(deg(Pi/4));
45
```

A function with two arguments:

```
> sum := << (x, y) -> x + y >>;

> print(sum(1, 2));
3
```

3.16 Comments

You should always document the code you have written so that you and others will understand its meaning 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 the `long comment` is started with the token sequence #/ and ends with the closing /# token sequence[3].

```
> #/ 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 (or with AgenaEdit that is shipped with the interpreter) 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 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 Windows. Then you can execute it at the Agena prompt by typing:

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

in UNIX or

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

in Windows. Note the forward slashes used in Agena for Windows.

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
```

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

in Windows. See Appendix A4 for further command-line switches.

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, 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 vector storing numbers, strings, booleans, and all other data types except **null** in sequential order
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 modifying 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, 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 7.7; also linked lists have been implemented using tables, see Chapter 7.6.

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 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`	

Table 2: Examples for valid and invalid names

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

```
abs and arccos arcsin arctan as assigned atendof bottom break by bye
case char clear cls copy cos cosh dec delete dict do elif else end
entier enum esac even exp fail false fi filled first finite for from
gethigh getlow global if imag in inc insert int intersect into is join
keys last left ln lngamma local lower minus nargs not od of or pop proc
qsadd real replace restart return right sadd seq sethigh setlow shift si
sign sin sinh size skip split sqrt subset tan tanh then to top trim
true type typeof unassigned union unique upper values while xor xsubset

boolean complex infinity lightuserdata null number pair procedure
sequence set string table thread undefined userdata
```

The following symbols denote other tokens:

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

4.2 Assignment

Values can be assigned to names in the following fashions:

$$name := value$$
$$name_1, name_2, \cdots, name_k := value_1, value_2, \cdots, value_k$$
$$name_1, name_2, \cdots, name_k \text{ -> } value$$

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 `->` operator.

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
```

4.3 Enumeration

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

enum *name*$_1$ [, *name*$_2$, $\cdots$]
enum *name*$_1$ [, *name*$_2$, $\cdots$] **from** *value*

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:

clear *name*$_1$ [, *name*$_2$, $\cdots$, *name*$_k$]

```
> a := 1;

> clear a;

> a:
null
```

which also performs a garbage collection useful if large structures shall be 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.

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
and
< > <= >= = == <> :: :-
in subset xsubset union minus intersect atendof
& :
+ - || ^^ split
* / % \ && shift *% /% +% -%
not - (unary minus)
^ **
! and all unary operators including ~~
```

As usual, you can use parentheses to change the precedence of an expression. The concatenation (`&`), exponentiation (`^`, `**`) and pair (`:`) 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 plus sign.

You may optionally include one or more single quotes *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 with 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`, or `t`, then the number is multiplied with 1,000, 1,000,000, 1,000,000,000, 1,000,000,000,000 or respectively.

```
> 2k:
2000

> 1M:
1048576

> 12D:
144
```

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

System	Syntax	Examples
binary	0b<binary number> or 0B<binary number>	0b10 = decimal 2
octal	0o<octal number> or 0O<octal number>	0b10 = decimal 9
hexadecimal	0x<hexadecimal number> or 0X<hexadecimal number>	0xa = decimal 10

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 because it provides arbitrary precision arithmetic. See Chapter 7.9 for more information.

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, faster than ^	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

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 shows 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 **readlib** or **with**[4] functions to activate the package before using these functions.

Unary operators[5] 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)	Arc 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`
roundf(x, d)	Rounds the real value x to the d-th digit	Base	`roundf(` `  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..`
sadd([· · ·])	Sum	Kernel	`sadd([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\` operation	`7 && 2 » 2`
||	Bitwise \`or\` operation	`1 || 2 » 3`
^^	Bitwise \`exclusive-or\` operation	`7 ^^ 2 » 5`
~~	Bitwise complementary operation	`~~7 » -8`

[4] Check the **with** function which also provides an easy way to define short names for package procedures.
[5] See Appendix A1 for a list of all unary operators.

Operator	Operation	Details / Example
shift	Bitwise shift	If the right-hand side is positive, the bits are shifted to the left (multiplication with 2), else they are shifted to the right (division by 2).

Table 5: Bitwise operators

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

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

The default is restored as follows:

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

You can query the higher and lower bits of a number with the **gethigh** and **getlow** operators and change them with the **sethigh** and **setlow** operators.

```
> a := gethigh(Pi):
1074340347

> b := getlow(Pi):
1413754136

> x := 0;

> x := sethigh(x, a):
3.1415920257568

> x := setlow(x, b):
3.1415926535898

> Pi = x:
true
```

4.6.3 Increment and Decrement

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[6] which are also around 10% faster:

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

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

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

```
> inc a;

> a:
3

> dec a;

> a:
2

> inc a, 2;

> a:
4

> dec a, 3;

> a:
1
```

4.6.4 Mathematical Constants

Agena features arithmetic constants mentioned in Appendix A9.

All mathematical functions 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`, `arcsin`, `arctan`, `cos`, `cosh`, `entier`, `exp`, `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`. 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

```
> 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 **approx** function might be of some help since it compares values approximately.

```
> approx(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** operator:

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

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 between them:

```
> '':
```

4.7.2 Substrings

You may obtain a specific character by passing its position in square brackets right behind the string name. If you use a negative index n, then the |n|-th character from the right end of the string is 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 as 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* is returned. If *end* is given too, then the substring starting at position *start* up to and including position *end* is 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[7] 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

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

[7] See also Appendix A7.

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: "
```

Likewise, a backslash is inserted 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
```

4.7.5 More on Strings

Instead of putting single or double quotes around a text, you may also use a back quote in front of the text, but not at its end. The string then automatically ends with one of the following tokens[8]:

```
<space> " , ~ [ ] { } ( ) ; : # ' = ? & % $ § \ ! ^ @ < > | \r \n \t
```

This also allows UNIX-style filenames to be entered using this short-cut method.

```
> `text:
text

> `/proglang/agena/lib/library.agn:
/proglang/agena/lib /library.agn
```

If you want to include double quotes in a string that is delimited by single quotes, backslashes may be omitted:

```
> '"Agena"':
"Agena"
```

And vice versa:

```
> "'Agena'":
'Agena'
```

[8] For the current settings of your Agena version see the bottom of the `agnconf.h` file in the `src` directory of the distribution.

4.7.6 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 is returned; otherwise **null** is 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 is returned; otherwise **null** is returned.
replace(s, p, r)	string	Replaces all patterns p in string s with substring r. If p is not in s, then s is 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 is 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 operator 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 operator looks for the first match of the substring and returns the position where the pattern starts with
		`instr( 'agena', '[aA]g', 1) » 1`
		
		`instr('agena', 'a', 'reverse') » 5`

Function	Functionality	Example
	respect to its left beginning. When searching from right to left, pattern mathing is not supported.	

atendof	Checks whether a string (right operand) ends in a substring (left operand). If true, the position is returned; otherwise **null** is 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 7.2.3.
	
	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 operator, too.
		`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 7.2.1 and Chapter 7.2.2. See also the descriptions of **strings.match** and **strings.gmatch**.

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

4.7.7 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
```

4.7.8 Patterns and Captures

Sometimes, just looking for a fixed pattern, e.g. a simple substring, in a string does not suffice. 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[9]:

- 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** operator 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** operator 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
```

[9] 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 7.2.3.

Character sets define user-defined classes determined by 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 with 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 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
```

Concerning recognising ligatures and umlauts, along with Latin letters, just use square brackets:

```
> strings.match('Selçuk, Turkey', '([çéöð%a]*)'):
Selçuk
```

Summary[10] 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
	%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
	%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 he class %w
	%x	hexadecimal digits 0 to 9, A to F, and a to f
	%X	anything not matching he class %x
	%z	an embedded zero, i.e. \0.
	%Z	anything not matching an embedded zero
Modifiers	+	one or more occurrences
	*	zero or more occurrences, returning the largest match
	-	zero or more occurrences, returning the smallest match
	?	zero or one occurrences

Table 9: Character classes and modifiers

[10] 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[11]	`[1] == [1]` `1 == 1`
<>	not equals	`1 <> 2`

Table 10: Relational operators

The logical operators **and**, **or**, and **xor** 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	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 **false**, **fail**, or **null** and if the second operand is neither **false**, **fail**, or **null**. Otherwise it returns **false**.	`true xor false» true`
		`true xor true » false`
		`false xor true » true`
		`1 xor null » 1`

[11] See Chapter 4.9.3.

Operator	Description	Examples

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

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 and sequences which are described later. Most operations on tables introduced in this chapter are also applicable to sets and sequences.

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:

[[*value*$_1$ [, *value*$_2$, $\cdots$]] **]**

```
> 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:

tablename[*key*]

```
> 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:

tablename[key_1][key_2][$\cdots$] *tablename*[key_1, key_2, $\cdots$]

```
> 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
```

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

tablename[*m* **to** *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 is returned. Table values with non-integer 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.

create table $name_1$ [, **table** $name_2, \cdots$]
$name_1$:= **[ ]**

```
> 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[12]:

insert *value₁* [, *value₂*, ···] **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** [13]statement works a little bit differently and removes all occurrences of a value from a table.

delete *value₁* [, *value₂*, ···] **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 the maximum number of entries has been reached. If so, each time it automatically enlarges the table 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 will be exceeded.

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

[13] dito.

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

create table *name*$_1$(*size*$_1$) [, **table** *name*$_2$(*size*$_2$), $\cdots$]

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:

[[*key*$_1$ $\sim$ *value*$_1$ [, *key*$_2$ $\sim$ *value*$_2$, $\cdots$]] **]**

```
> 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';
```

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

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

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

create {**table** | **dict**} *name₁*[(*size₁*)] [, {**table** | **dict**} *name₂*[(*size₂*)], ···]

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**.
A = B	Boolean	Checks whether two tables A, B, or two sets A, B, or two sequences A, B contain the same values regardless of the number of their occurrence; if B is a reference to A, then the result is also **true**.
A <> B	Boolean	Checks whether two sets/tables/sequences A, B do not contain the same values regardless of the number of their occurrence; if B is a reference to A, then the result is **false**.
A == B	Boolean	Checks whether two tables A, B, or two sets A, B, or two sequences A, B contain the same number of elements and whether all key~value pairs in the tables or entries in the sets or sequences are the same; if B is a reference to A, then the result is also **true**.
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.

Name	Return	Function
A **union** B	table, set, seq	Concatenates two tables, or two sets, or two sequences 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	Returns all values in two tables, two sets, or two sequences A, B that are included both in A and in B as a new structure.
A **minus** B	table, set, seq	Returns all the values in A that are not in B as a new structure.
copy A	table, set, seq	Creates a deep copy of the structure A, i.e. if A includes other tables, sets, pairs, or sequences, copies of these structures are built, too.
join A	string	Concatenates all strings in the table or sequence A.
size A	number	Returns the size of a table A, i.e. the actual number of key~value pairs in A. With sets and sequences, the number of items is returned.
sort(A)	table, seq	This function sorts table or sequence 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	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.
sadd A	number	Sums up all numeric table or sequence values. If the table or sequence is empty or contains no numeric values, **null** is returned. Sets are not supported.
qsadd A	number	Raises each value in a table or sequence to the power of 2 and sums up these powers. If the table or sequence is empty or contains no numeric values, **null** is returned. Sets are not supported.

Table 12: Table, set, and sequence operators

Here are some examples - try them with sets and sequences 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 set at the left hand side of the operator, it is written the same number of times to the resulting table.

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 set at the left hand side of the operator, it is written the same number of times to the resulting table.

The **unique** operator

- 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), e.g.:

Function	Description	Further detail
map(*f*, *o*)	Maps a function *f* onto all elements of structure *o*.	*f* may be an anonymous function, as well. See also **zip** in Chapter 7.1.
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.

Function	Description	Further detail
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*.	

Table 13: Basic table procedures

The **map** function is quite handy to apply a function to all elements of a table by one stroke:

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

Suppose we want to add a new entry 10 at position 3 of table c[14]:

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

> put(C, 3, 10)

> 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.

[14] **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.

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 Chapter 7 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 tables, sets, sequences, or pairs, 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, or sequences, all names to the left of an

-> operator will point to the very same structure to its right. This behaviour may be changed in a future version of Agena.

```
> 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)]
```

4.10 Sets

Sets are collections of unique items: numbers, strings, and any other data except **null**. Their syntax is:

{ [*item*$_1$ [, *item*$_2$, · · ·]] }

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[*element*]

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 not supported with sets.

insert *item*$_1$ [, *item*$_2$, $\cdots$] **into** *name*

delete *item*$_1$ [, *item*$_2$, $\cdots$] **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 *name*$_1$ [(*size*$_1$)] [, **set** *name*$_2$ [(*size*$_2$)], $\cdots$]

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 further entries.

Sets are useful in situations where the number of occurrences of a specific item or its position do 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** operator 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 occurrences of a unique element in a distribution, the **bags** package might be of interest, see Chapter 7.7.

The following operators work 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**.
A = B	Boolean	Checks whether two sets A, B contain the same values regardless of the number of their occurrence; 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 regardless of the number of their occurrence; 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.
copy A	set	Creates a deep copy of the set A, i.e. if A includes other tables, sets, pairs, or sequences, 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.

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.

Sequences store items in sequential order. Like in 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. Other types of indexes 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

> a[2]:
1
```

If the index is larger than the current size of the sequence, an error is returned[15].

```
> a[5]:
Error, line 1: index out of range
```

Sublists of sequences can be determined with the following syntax:

seqname[*m* **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 is issued.

[15] The error message can be avoided by defining an appropriate metamethod.

```
> 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:

```
> 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 **false** or **true**, respectively.

```
> a :: sequence:
false

> a :- sequence:
true
```

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.

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** operator 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 work on sequences:

Name	Description	Example
=	Equality check the Cantor way	`a = b`
==	Strict equality check	`a == b`
<>	Inequality check the Cantor way	`a <> b`
::	Type check operator	`a :: sequence` `a :: 'usertype'`
:-	Negation of type check operation	`a :- sequence` `a :- 'usertype'`
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`
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`
getentry	Returns entries without issuing an error if a given index does not exist.	`getentry(a, 1, 3)`

Name	Description	Example
in	Checks whether an element is stored in the sequence, returns **true** or **false**.	`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 7.1.	`unpack(a)`
map	Maps a function on all elements of a sequence.	`map(<< x -> x^2 >>, seq(1, 2, 3))`
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 another 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 another 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 another 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 6.25, if you have to conduct a lot of insertions and/or deletions in a data structure.

4.12 Stack Programming

Sequences and table arrays can be used to implement stacks, and besides **insert**, two efficient statements are available to remove an item from the bottom of the stack or from the top of the stack:

pop bottom from *name*
pop top from *name*

Both statements 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 change the stack, i.e. the sequence or table.

```
> 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)
```

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, seq 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 already 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:
false

> p :: 'duo':
true
```

The only other operators besides **left** and **right** that work on pairs are equality (= and ==), 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** operator if you want to create `independent` pairs.

Summary:

Name	Description	Example
=, ==	Equality checks (same functionality)	`a = b`
<>	Inequality check	`a <> b`
::	Type check operator	`a :: pair` `a :: 'udeftype'`
:-	Negation of type check operation	`a :- pair` `a :- 'udeftype'`
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)`

Name	Description	Example
getmetatable	Returns the metatable stored to a pair.	`getmetatable(p)`

Table 16: Operators and functions applicable to pairs

4.15 Exploring the Internals of Structures

If you would like to know how a table, set, sequence, or pair is represented internally, please have a look at the **environ.attrib** function explained in Chapter 7.20. It might help when debugging code.

It 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.

4.16 Other types

For threads, userdata, and lightuserdata please refer to the Lua 5.1 documentation.

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

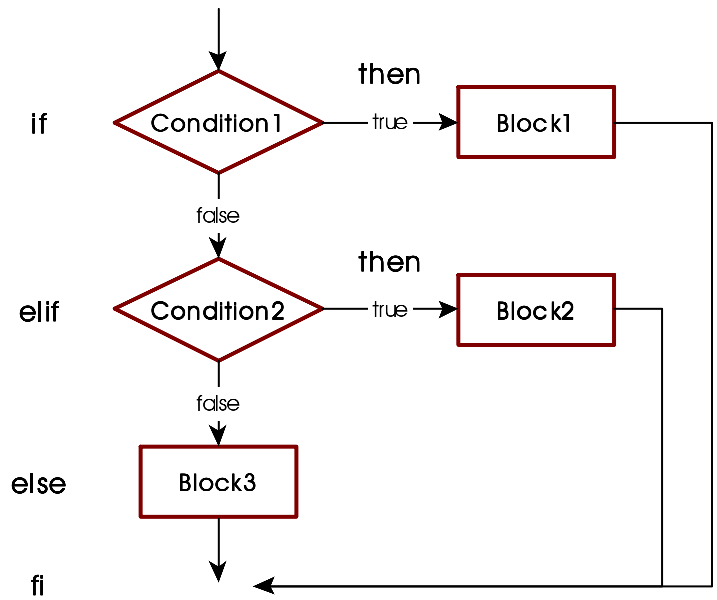
The **if** statement checks a condition and selects one statement from many listed. Its syntax is as follows:

if *condition$_1$* **then**
  statements$_1$
[**elif** *condition$_2$* **then**
  statements$_2$]
[**else**
  statements$_3$]
fi

The condition may always evaluate to one of the Boolean values **true**, **false**, or **fail**, **or to any other value** .

The **elif** and **else** clauses are optional. While more than one **elif** clause can be given, only one **else** clause is accepted. An **if** statement may include one or more **elif** clauses and no **else** clause.

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 any condition results to **false**, **fail**, or **null**, the **else** clause is executed if present, otherwise Agena proceeds with the next statement following the **if** statement.



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
```

5.1.2 is Operator

The **is** operator checks a condition and returns the respective expression.

is *condition* **then** *expression*$_1$ **else** *expression*$_2$ **si**

This means that the result is *expression$_1$* if *condition* is **true** or any other value except **false**, **fail**, or **null**; and *expression$_2$* otherwise.

Example:

```
> x := is 1 = 1 then true else false si:
true
```

which is the same as:

```
> if 1 = 1 then
>    x := true
> else
>    x := false
> fi;
```

The **is** 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 string indices, etc. You may nest **is** operators.

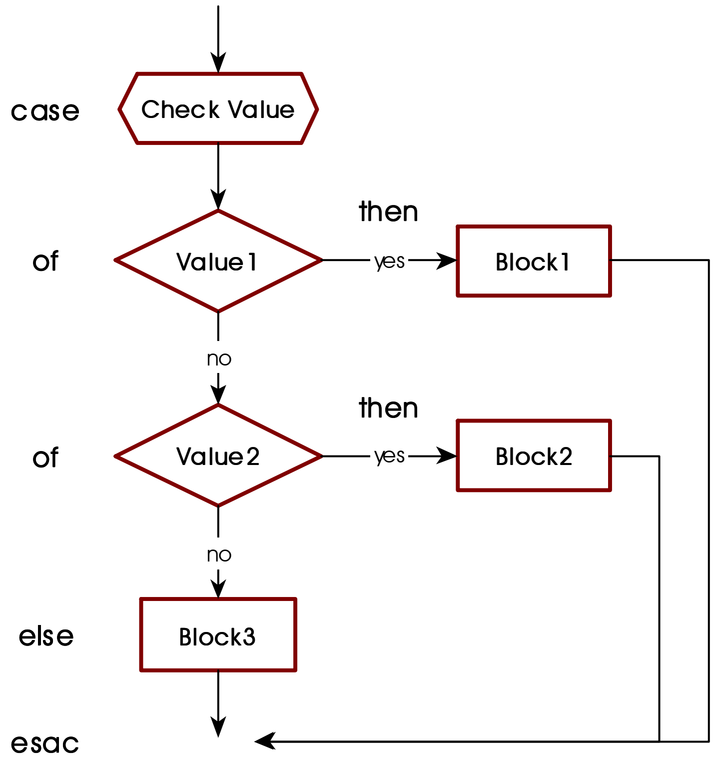
5.1.3 case Statement

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

case *name*
  of *value$_{11}$* [, *value$_{12}$*, $\cdots$] **then** *statements$_1$*
  [**of** *value$_{21}$* [, *value$_{22}$*, $\cdots$] **then** *statements$_2$*]
  [**of** $\cdots$]
  [**else** *statements$_k$*]
esac

```
> a := 'k';

> case a
>    of 'a', 'e', 'i', 'o', 'u', 'y' then result := 'vowel'
>    else result := 'consonant'
> esac;

> result:
consonant
```

You can add as many **if/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.



5.2 Loops

Agena has two 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.

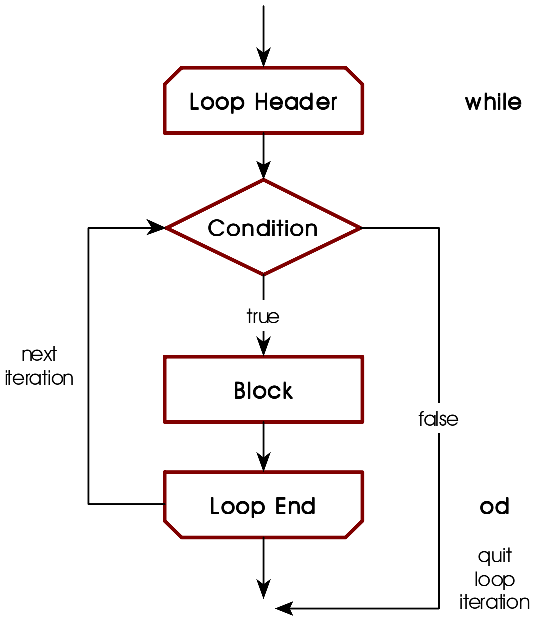
If the condition is **false**, **fail** or **null**, no further iteration is done and control returns to the statement following right after the loop body.

If the condition is **false**, **fail**, or **null** from the start, the loop is not executed at all.

while *condition* **do**   *statements* **od**

Thus the programme flow is as shown in the diagram.

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
```

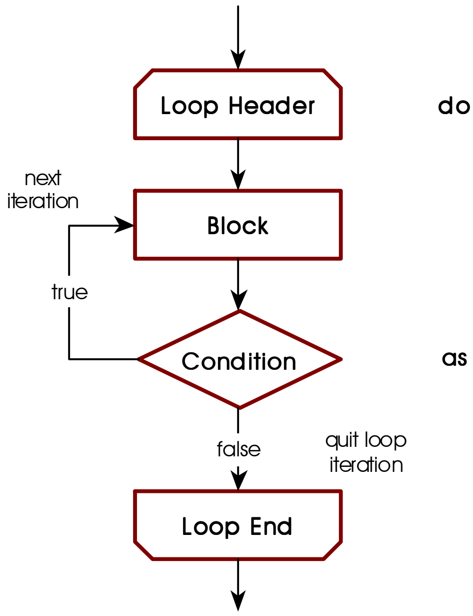
The following loop will never be executed since the condition is **false**:

```
> while false do
>    print('never printed')
> od;
```

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

do
   statements
as *condition*

```
> c := 0;

> do
>    inc c
> as c < 10;

> c:
10
```



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 5.2.9 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.

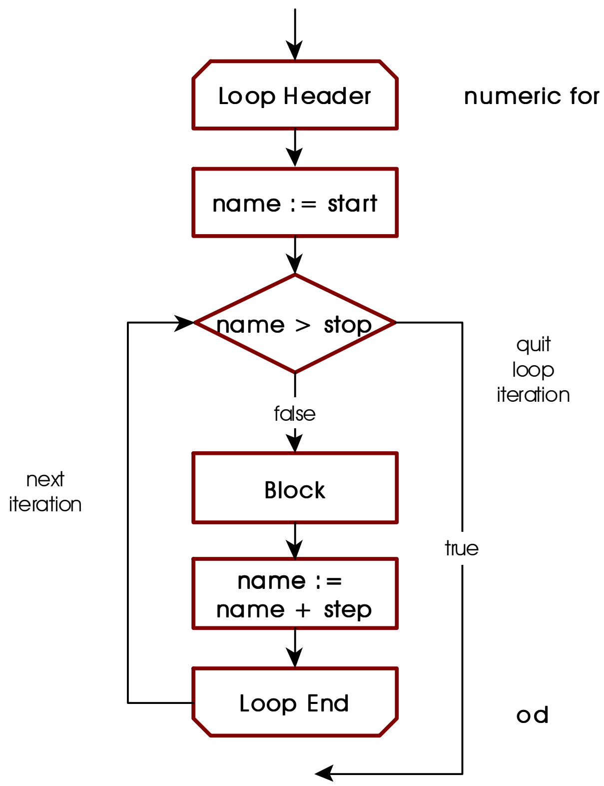
5.2.2 for/to Loops

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 numeric value of *start*. *name* is called the *control* or *loop variable*. If *start* is not given, the start value is $+1$.



It then checks whether *start* $<=$ *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. If *step* is not given, the control variable is always incremented by $+1$.

```
> 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** extension. Note that within procedures, the loop control variable is automatically declared local, while on the interactive level, it is global.

```
> for i to 1e300 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 its stop value, 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[16] within the loop, or if a **for/while** loop is terminated because the **while** condition evaluated to **false**, 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.

Loops can also count backwards if the step size is negative:

```
> 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 Kahan summation if the step size is non-integral, e.g. 0.1, -0.01. This prevents round-off errors and 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.

If the step size is an integer, e.g. 1000, -1., then Agena does not use advanced precision to ensure maximum speed.

5.2.3 for/in Loops over Tables

are used to traverse tables, strings, sets, and sequences, and also iterate 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.

---

[16] See Chapter 5.2.8 for more information in the **break** statement.

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 keyword **keys** in front of the control variable, the loop iterates only on 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** operator 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 implement and linked list (see Chapter 6.25).

5.2.4 for/in Loops over Sequences

All of the features explained in the last subchapter are applicable to sequences, as well.

5.2.5 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 := result & 'V';
>          inc v
>       of 'å', 'æ', 'ø', 'ö' then
>          result := result & 'L';
>          inc l
>       else
>          result := result & 'C'
>          inc c
>    esac
> od;
```

```
> print(result, v & ' vowels', l & ' ligatures', c & ' consonants');
LCCVC       1 vowels       1 ligatures        3 consonants
```

5.2.6 for/in Loops over Sets

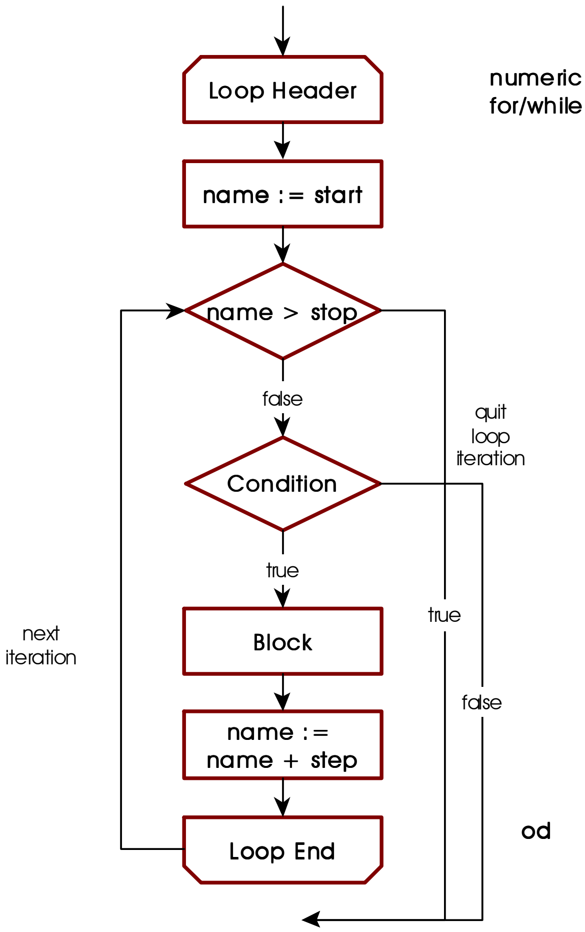
All **for** loop variations are supported with sets, as well. The only useful one, however, is the following:

```
> 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.

5.2.7 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** which quits the **for/in** iteration. See Chapter 6 which describes procedures in detail.




```
> double := proc(state, n) is
>    if n < state then
>       inc n;
>       return n, 2*n
>   else
>       return null
>    fi
> end;

> for i, j in double, 5, 0 do
>    print(i, j)
> od
1       2
2       4
3       6
4       8
5       10
```

5.2.8 for/while 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**.

for *i* [**from** *a*] **to** *b* [**by** *step*] **while** *condition* **do** *statements* **od**
for [*key*,] *value* **in** *struct* **while** *condition* **do** *statements* **od**
for keys *key* **in** *struct* **while** *condition* **do** *statements* **od**
for [*key*,] *value* **in** *string* **while** *condition* **do** *statements* **od**
for keys *key* **in** *string* **while** *condition* **do** *statements* **od**

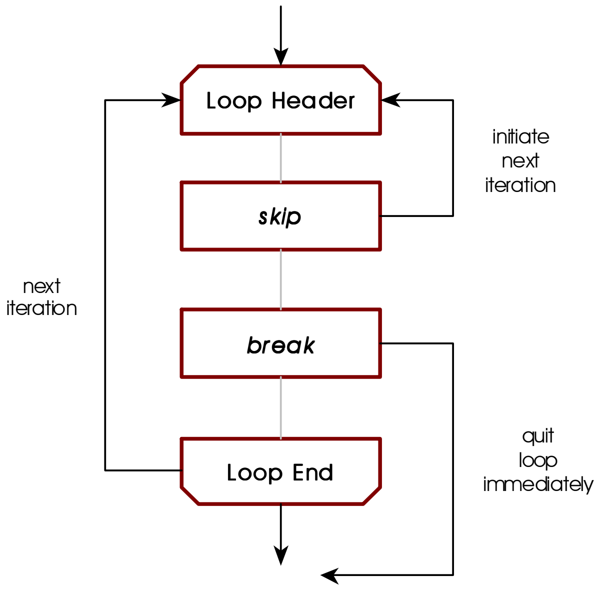
An example:

```
> 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: 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*.

5.2.9 Loop Interruption

Agena features two statements to manipulate loop execution. Both are applicable to all loop types.



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
```

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

Writing procedures in Agena is quite simple:

procname := **proc**([*par*$_1$ [::*type*$_1$] [, *par*$_2$ [::*type*$_2$], $\cdots$]]) [:: *returntype*] **is**
  [**local** *name*$_1$ [, *name*$_2$, $\cdots$]];
  statements
end

All the values that a procedure shall process are given as *parameters par*$_1$, etc. A function may have no, one, or more parameters. A parameter may be succeeded by the name of a type (see Chapter 6.7). If a type is given right after the parameter list, Agena checks whether the return of the procedure is of the given return type, 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 used 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 is returned using the **return** keyword which may be put anywhere in the procedure body, and which also immediately terminates further execution of the procedure.

return [*value* [, *value*$_2$, $\cdots$]]

As you can see, you may not only return a single result, but also multiple ones, or none at all.

Furthermore, a procedure does not return anything - not even the **null** value -

- if no **return** statement is given at all,
- if no arguments are passed to the **return** statement.

The following procedure computes the factorial of an integer[17]:

```
> 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 called using the syntax:

funcname(**[***arg₁* [, *arg₂*, · · ·]**]**)

```
> fact(4):
24
```

where the first parameter is replaced by the first argument arg_1, the second parameter is substituted with arg_2, etc.

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 it 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:

[17] The library function **fact** is much faster.

local *name*$_1$ [, *name*$_2$, $\cdots$]
local *name*$_1$ [, *name*$_2$, $\cdots$] := *value*$_1$ [, *value*$_2$, $\cdots$]
local *name*$_1$ [, *name*$_2$, $\cdots$] -> *value*
local enum *name*$_1$ [, *name*$_2$, $\cdots$] [**from** *value*]

In the first form, *name*$_1$, etc. are declared local.

In the second and third form, *name*$_1$, etc. are declared local and, as opposed to the first form, followed by initial assignments of values to these names.

In the last form, *name*$_1$, etc. are declared local with a subsequent enumeration of those names, i.e. assignment of ascending positive integers to these names.

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 cases.

```
> 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 so it has no value at the interactive level.

```
> result:
null
```

There is a shortcut to create local structures - tables, sets, and sequences:

create local <structure> *name*$_1$ [, <structure> *name*$_2$, $\cdots$]

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 indicated with the **global** keyword. This is optional, however, and only serves documentary purposes.

```
> fact := proc(n) is
>    global _EnvMoreInfo;
>    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 _EnvMoreInfo then print('Overflow !') fi;
>          break
>       fi
>    od;
>    return result
> end;
```

We must 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 assign new values to 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 avoid using a global variable to get a warning message by passing an optional argument instead.

```
> 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
```

The option should 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.

```
> varadd := proc(?) is
>    local result := 0;
>    for i to size varargs do
>       inc result, varargs[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, varargs[i]
>    od;
>    return result
> end;

> varadd(1, 2, 3, 4, 5):
15
```

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[18].

```
> xsqrt(-2, domain = 'complex'):
1.4142135623731*I
```

6.6 Passing Options in any Order

We can combine the varargs facility with the usage of 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(varargs[i])
>          of 'bailout' then
>             bailout := right(varargs[i]);
>          of 'iterations' then
>             iterations := right(varargs[i]);
>          else
>             print 'unknown option'
>       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
```

[18] If you need to conduct a Boolean equality operation in a function call, such like `f(a=b)`, use the **isequal** function, like `f(isequal(a, b))`.

Again, the single quotes around the name of the option (left-hand side of the pair) can be spared by using the = token which converts the given 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** function which may save up to 20 % of code. You might see Chapter 7.1 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 checks for a basic or user-defined type.
3. The :: operator evaluates a value for a given type or user-defined type,
4. The :- operator checks whether a value is not of a given type or user-defined type,
5. A basic or user-defined type can be optionally specified in the parameter list of a procedure by means of the preceding `::` token so that it will be checked at procedure invocation.
6. The type of return of a procedure may be given right after the parameter list.

The following standard types are available in Agena:

```
boolean, complex, lightuserdata, null, number, pair, procedure,
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 that returns the type of a value as a string:

type(*value*)

```
> type(1):
number

> type(1) = number:
true
```

The only exception to the above is when checking for the type of anything evaluating to **null**. In this case, put the **null** constant into 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, as shown in the example above.

value :: *typename* *value* :- *typename*

Examples:

```
> 1 :: number:
true

> '1' :- number:
true
```

In case of user-defined types, the type name must always be a string put into quotes. See Chapter 6.12 for more information.

6.8 Error Handling

The **error** function immediately terminates execution of the procedure, and prints an error message if given.

error('*error string*')

```
> fact := proc(n) is
>    if n :- number then
>       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
```

You may optionally 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 issues an error. Agena will also return an error if the argument is not given:

```
> fact()
Error in stdin:
   missing argument #1 (type number expected).
```

Agena can check whether all returns of a procedure are of a given single 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
```

There are three other functions for error handling:

assume checks a Boolean relation. In case the relation is valid, it returns **true** and all other arguments given. In case of an invalid relation, it terminates execution of the procedure and prints an error message. The second argument to **assume** is optional; if not given, the text `assumption failed` is returned with invalid relations.

assume(relation [, 'error string'])

```
> 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
```

protect traps any error, but does not terminate a function call. In case of no errors, it returns all results of the call. In case of 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** is always **null**.

protect accepts the name of the function f to be executed as its first argument, and all arguments a, b, $\cdots$ of f as optional arguments:

protect(f [, a [, b, $\cdots$]]**)**

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;
```

To call iszero in protected mode, enter:

```
> 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
```

Note that **protect** does not directly work with operators, instead you may include a call to an operator in a new function:

```
> mycopy := proc(x) is
>    return copy(x)
```

```
> end;

> protect(mycopy, 1:1) = lasterror:
true
```

Finally, **argerror** is a little bit smarter than **error** for it automatically returns the type of an argument actually passed to a procedure.

```
> a := 1;

> if a :- string then
>    argerror(a, 'myproc', 'expected a string')
> fi
Error in `myproc`: expected a string, got number.
```

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 or table:

```
> seq(f()):
seq(1, 2)
```

Sometimes a procedure shall only 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.

ops(index, arg_1 [, $\text{arg}_2, \cdots$] **)**

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 := map(<< x -> x^2 >>, t);
>    return unpack(result)
> end;

> squared([1, 2, 3, 4]):
1       4       9       16
```

Optionally, **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 are determined.

unpack(structure [, beginning [, end]] **)**

```
> squared := proc(t::table, ?) is
>    local result := map(<< x -> x^2 >>, t);
>    return unpack(result, unpack(varargs))
> 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 new procedures. As an example, the function `polygen`

```
> polygen := proc(?) is
>    local s := seq(unpack(varargs));
>    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 to evaluate 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.

<< [(] [*par*$_1$ [:: *type*$_1$] [, *par*$_2$ [:: *type*$_2$], $\cdots$]] [)] -> expr >>

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, even if you specify types.

```
> isInteger := << x -> int(x) = x >>;

> isInteger(1):
true

> isInteger(1.5):
false
```

Passing optional arguments using the ? notation is supported. In this case, use the **varargs** table as described above.

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 has been set, 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, you may use the double colon notation in its parameter list.

Suppose you have defined a type called `triple`:

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

> settype(t, 'triple')
```

```
> sum := proc(x::triple) is
>    return sadd(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.

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** loops;
5. **is** and **end** - in procedures;
6. **scope** and **epocs** - in **scope** blocks (see below).

As an example, variables declared as local within procedures are only visible to the block in which they have been defined. Especially, they cannot be accessed from outside the procedure in which they are hosted.

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^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 other values. You might want to limit this access, for example if one of your procedures offers services on the Internet, or 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 not featuring the **os** package:

```
> environ.setfenv(curdir, _newG);

> curdir():
Error in stdin, at line 2:
   attempt to index global `os` (a null value)

Stack traceback:
   stdin, at line 2, at line 1
```

Example 2: Give access only the specific functions

```
> 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 7.20 (**environ.getfenv**, **environ.setfenv**, **environ.isselfref**) for further features.

6.16 Altering the Environment at Run-Time

Besides using a special environment (see the subchapter above), 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 her/his 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;
>    if n = 0 then
>       return 1
>    else
>       return 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 with Function

The **with** 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 prepend or append the location of the directory containing your new package to **libname**, or execute Agena in the directory containing your package. You may do this by adding the following line into your personal Agena initialisation file (see Chapter A6), assuming that the `helpers.agn` file has been stored to the folder `d:/agena/helpers`.

```
libname := 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.

```
> with '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;
```

```
> with '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;

> with '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 **with** 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 run')
> end;
```

Of course, you must create a \`*packagename*.**aux**\` table before defining the initialisation function.

6.18 Remember Tables

Agena features remember tables which if present hold the results of previous calls to Agena or API C procedures or contain a list of predefined results, or both. If a function is called again with the same argument or the same arguments, then the corresponding result is returned from the table, and the procedure body is not executed. 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, they must explicitly be created with the **rtable.rinit** function and optionally filled with default values with the **rtable.rset** 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.

Two examples: We want to define a function f(x) = x with f(0) = undefined.

First the function is defined:

```
> f := << x -> x >>;
```

Only after the function has been created, the rtable (short for remember table) can be set up. The **rtable.remember** function can be used to initialise rtables, explicitly set predefined values to them, and add further values later in a session.

```
> with('rtable');
defaults, rdelete, remember, rget, rinit, rmode, roinit, rset

> remember(f, [0 ~ undefined]);
```

The rtable has now been created and a default entry included in 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 rtable is destroyed, so you may have to initialise it again.

Fibonacci numbers can be implemented recursively and run with astonishing speed using rtables.

```
> fib := proc(n) is
>    assume(n >= 0);
>    return fib(n-2) + fib(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, [0~1, 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 7.22 for more details on their use.

6.18.2 Read-Only Remember Tables

If you do not want that a function updates 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.rset** function or a utility that is based on **rtable.rset**, 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 int(x) = x 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**.

6.18.3 Functions for Administering Remember Tables

For completeness, all basic functions that work on remember tables are the following:

Procedure	Details
rtable.rget(f)	Returns the remember table of function `f`.
rtable.rinit(f)	Initialises a standard remember table for the function `f`.
rtable.roinit(f)	Initialises a read-only remember table for the function `f`.
rtable.rset(f, **[**arguments**]**, **[**returns**])**	Adds function argument(s) and the corresponding return(s) to the remember table of procedure `f`.
rtable.rdelete(f)	Deletes the remember table of function `f` entirely. If you want to use a new remember table with the function, you have to initialise it with **rtable.rinit** or **rtable.roint** again.
rtable.rmode(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 17: 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, and pairs.

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[19].

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.

At first, lets store a complex value $z = x + yi$ to a sequence of size 2. The real part is saved as the first value, the imaginary part at 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
```

[19] For performance reasons, complex arithmetic has been built directly into the Agena kernel.

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), 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 is z[2]<0 then z[1]&z[2]&'i' else z[1]&'+'&z[2]&'i' si;
> 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 with 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 works as follows:

- If the structure is a table, then the metamethod is called if the call to an indexed name results to **null**.
- If the structure is a set, then the metamethod is called if the call to an indexed name results to **false**.
- If the structure is a sequence, then the metamethod is called if the call to an indexed name 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 respective 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 enter 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 be capable to 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
```

Using the **__writeindex** metamethod, it is quite easy to write-protect structures.

```
> readonly_mt := [
>    '__writeindex' ~
>       proc(t, k, v) is error('Error, structure is read-only.') end
> ]
```

A constructor 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 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]
```

To protect metatables from tampering, use the **__metatable** method and set it to any value except **null**.

```
> readonly_mt := [
>    '__writeindex' ~
>       proc(t, k, v) is error('Error, table is read-only') end,
>    '__metatable' ~ false
> ]

> 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
```

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 situated at another part of the memory.

```
> environ.pointer(s):
008A4188
```

Since the first set 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, or 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;
>    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 internal variables which can survive and keep their values even after the call to the procedure completed.

So with a successive call to the same procedure, it can access these values and use them in the current call again.

Let us define an iterator function that successively returns an element of a table:

```
> 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 returns the closure as a function which we assign to the name `iterator`. The `iterator` function remembers its state and can be called like `normal` functions:

```
> iterator := traverse(['a', 'b', 'c']);

> iterator():
a
```

What happened ? The call to `traverse` with the table `['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. With the first call to `iterate`, count was incremented from `0` to `1`, followed by the return of the first element in the table.

```
> iterator():
b

> iterator():
c
```

Since the table has no more elements left (count = 4), it now returns **null**.

```
> iterator():
null
```

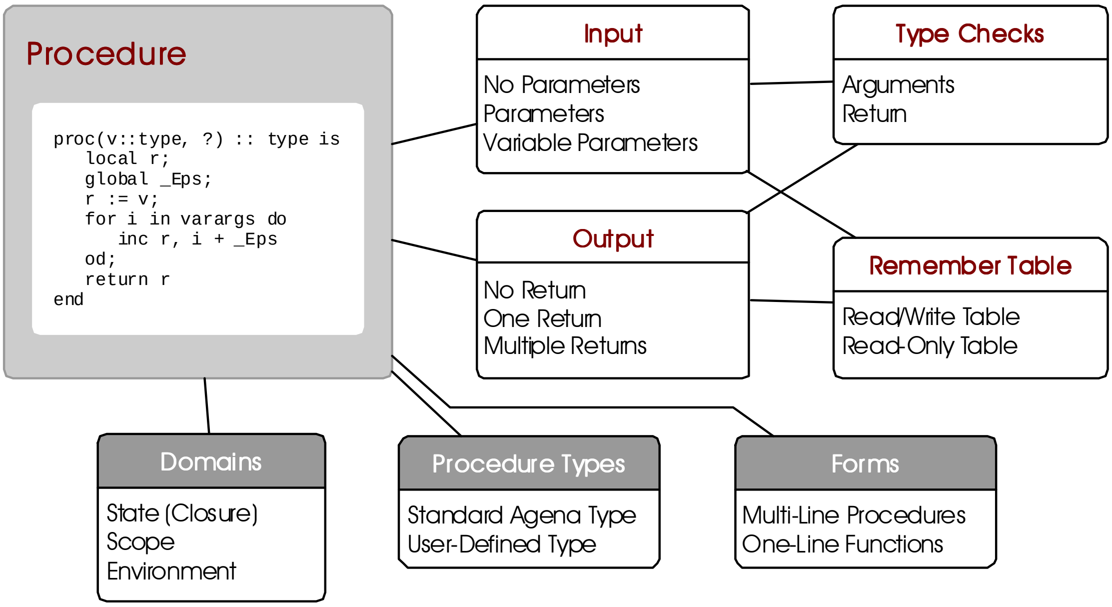
You can define more than one closure with a factory at the same time, each being completely independent from the others:

```
> iterator2 := traverse(['a', 'b', 'c']);

> iterator2():
a

> iterator2():
b
```

```
> iterator3 := traverse(['a', 'b', 'c']);

> iterator3():
a
```

6.23 Summary on Procedures

The following diagram tries to summarise all features of a procedure.



6.24 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 using **io.close**.

6.24.1 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 10 characters:

```
> io.read(fh, 10):
 for Agena
```

Close the file:

```
> io.close(fh):
true
```

Besides file handles, many IO 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
```

6.24.2 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 io 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 above statements produce the file contents:

```
I am a text.Me too.
3.1415926535898
```

In the next example we append text to the 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**[20]. Using the `'w'` switch would replace the text already existing with the new one. See Chapter 7.13 for further options accepted by **io.open**.

The file looks like this:

```
I am a text.Me too.
3.1415926535898
20
```

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 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, Windows, or OS/2, just open the text file in binary mode. This avoids carriage return control codes to be added to the file with each line break.

See Chapter 7.13 for a description of all **io** package functions.

6.24.3 Keyboard Interaction

The **io.read** function allows to enter values interactively via the keyboard when called with no arguments. 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

> a:
10

> type(a):
string
```

[20] See Chapter 7.13 for further options accepted by **io.open**.

```
> 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 number. 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 18: Functions to read the keyboard

6.24.4 Default Input, Output, and Error Streams

Agena, enherited from Lua, 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
```

6.24.5 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 also try to access them. See **io.lock** and **io.unlock** in Chapter 7.13 for further information.

6.24.6 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**.

6.24.7 CSV Files

Comma-separated value files can be read conveniently by **utils.readcsv**. This function provides various options to further process the data being read. See Chapter 7.25 for further details.

6.24.8 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 7.25 offers further information on how to do this.

6.24.9 dBASE III Files

The xbase package can read and write dBASE III-compatible files.

6.25 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.

Also iterating a table with the **for/in** statement does not ensure that the keys are traversed in ascending order[21].

[21] See **skycrane.iterate** .

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. Just take a look at the examples at the end of this subchapter.

The benefit of using linked list in these situations is 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 the list 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 use:

```
> l := list;

> while l do
>    if l.data = -1 then
>       l.next := ['data' ~ -1.5, 'next' ~ l.next];
>       break
>    fi;
>    l := l.next
> od;

> 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.

Using the **llist** package is easy. First initialise it,

```
> readlib '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)
```

Chapter Seven

Standard Libraries

7 Standard 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,
- 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.

7.1 Basic Functions

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.

Summary of functions:

Checks

abs, **assigned**, **assume**, **filled**, **has**, **isequal**, **rawequal**, **whereis**.

Extraction

bottom, columns, duplicates, getentry, left, max, min, next, ops, rawget, 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, nonneg, notisnegint, notisposint, settype, type, typeof.

Counting

countitems, size.

Data Manipulation

alternate, augment, getbit, map, purge, put, rawset, remove, select, selectremove, setbit, sort, sorted, subs, toseq, toset, totable, zip.

Data Generation

dimension, nseq.

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**.

`abs (x)`

If `x` is a number, the **abs** operator will return the absolute value of `x`. Complex numbers are supported.

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` is or evaluates to **null**, else returns `y`.

`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 type of `x`. Thus it returns the error message: 'Error in `procname`: `message`, got <type of `x`>.'.

The function is written in the Agena language and included in the `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 or sequences `obj1`, `obj2` together horizontally. The arguments must either be tables only, or sequences only. The tables or sequences all must have the same size. The type of return is determined my the type of the arguments.

The function is written in the Agena language and included in the `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. 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.

bintersect (obj1, obj2 [, option])

Returns all values of table or sequence `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 tables or sequences, this function is much faster than the **intersect** operator.

The function is written in the Agena language and included in the `library.agn` file.

See also: **bisequal**, **bminus**.

bisequal (obj1, obj2 [, option])

Determines whether the tables `obj1` and `obj2` or sequences `obj1` and `obj2` contain the same values. The function performs a binary search in `obj2` for each value in `obj1`. If no option is given (any value), `obj2` is sorted before starting the search. If you pass an option of any type then `obj2` should already have been sorted, for no correct results would be returned otherwise.

With larger tables or sequences, this function is much faster than the = operator.

The function is written in the Agena language and included in the `library.agn` file.

See also: **bintersect**, **bisminus**.

bminus (obj1, obj2 [, option])

Returns all values of table or sequence `obj1` that are not 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 the option then `obj2` should already have been sorted, for no correct results would be returned otherwise.

With larger tables or sequences, this function is much faster than the **minus** operator.

The function is written in the Agena language and included in the `library.agn` file.

See also: **bintersect**, **bisequal**.

bottom (obj)

With the table array or sequence `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.

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=';'` 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 which will have to check its arguments `obj`.

Its second argument `obj` - a table - represents the arguments to be checked passed to `procname`.

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. an option name. 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 is issued. But if the very last argument is the Boolean value **true**, no error is 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 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, or sequence, 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 the Agena language and included in the `library.agn` file.

See also: **type**.

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 from the two-dimensional table or sequence `obj`. The type of return is determined by the type of `obj` and is either a table/sequence of tables/sequences if the option `'structure'` is given, or a multiple return of tables or sequences.

The function is written in the Agena language and included in the `library.agn` file.

See also: **linalg.column**, **utils.readscv**.

copy (obj)

The operator 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.

The type of return is determined by the type of `obj`.

The operator also treats cycles (structures that directly or indirectly reference to themselves), correctly.

countitems (item, obj)
countitems (f, obj [, ···])

In the first form, counts the number of occurrences of an `item` in the structure (table, set, or sequence) `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.

dimension (a:b [, c:d] [, init])

Creates a 1-dimensional sparse table or a 2-dimensional sparse table with arbitrary index ranges (of type pair) `a:b` and `c:d`. If the last argument is not a pair, it is used as an initialiser for all elements, otherwise all elements default to **null**.

If the initialiser is a structure, i.e. table, set, sequence or pair, then individual copies of the initialiser are created to avoid referencing to the same structure.

duplicates (obj [, option])

Returns all the values that are stored more than once to the given table or sequence `obj`, and returns them in a table or sequence. Each duplicate is 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 function is written in the Agena language and included in the `library.agn` file.

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**.

_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, or sequence `obj` contains at least one item and returns **true** if so; otherwise it returns **false**.

getbit (x, pos)

Checks for the bit at position `pos` $\in$ [1, 31] in the integer `x`, and either returns **true** or **false**.

See also: **setbit**.

getentry (obj [, k_1, ···, k_n])

Returns the entry obj[k_1, ···, k_n] from the table or sequence `obj` without issuing an error if one of the given indices k_i (second to last argument) does not exist. It conducts a raw access and thus does not invoke any metamethods.

If obj[k_1, ···, k_n] does not exist, **null** is returned. If only `obj` is given, it is simply returned.

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**.

gettype (obj)

Returns the type - set with **settype** - of a function, sequence, set, or pair `obj` as a string. If no user-defined type has been set, or any other data type has been passed, **null** is returned.

See also: **settype**, **typeof**.

has (obj, x)

Checks whether the structure `obj` (a table, set, sequence, or pair) contains element `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, 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: **in**, **recurse**.

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 the Agena language and included in the `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**.

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**, **innonneg**, **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: **notisposint**.

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**.

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))()
```

See also: **strings.dump**.

map (f, obj [, ···])

This operator maps a function `f` to all the values in table, set, sequence, 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]
```

See also: **nseq**, **remove**, **select**, **subs**, **zip**.

max (obj [, 'sorted'])

Returns the maximum of all numeric values in table or sequence `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.

See also: **min**, **math.max**, **stats.minmax**.

min (obj [, 'sorted'])

Returns the minimum of all numeric values in table or sequence `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.

See also: max, math.min, stats.minmax.

next (obj [, index])

Allows a programme to traverse all fields of a table or all items of a set or sequence `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 or sequences, **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.

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 non-existent field in the structure. With tables, you may however modify existing fields. In particular, you may clear existing table fields.

See also: skycrane.iterate.

notisnegint (···)

Checks whether all of its arguments are positive integers or zero and returns **true** or **false**. If at least one of its arguments is not a number, the function returns **fail**.

See also: notisposint.

notisposint (···)

Checks whether all of its arguments are non-positive integers and returns **true** or **false**. If at least one of its arguments is not a number, the function returns **fail**.

See also: isposint.

nseq (a, b [, step])

nseq (f, a, b [, step [, ···]])

In the first form, creates a sequence **seq(**`a`, `a+step`, ···, `b-step`, b**)**, with `a`, `b`, and `step` being numbers. The step size is 1 if `step` - a number - is not given.

In the second form, the function returns a sequence **seq(**1~`f`(a), 2~`f`(`a+step`), ···, (`(b-a)*`1/`step`+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 `step`.
The function uses the Kahan summation algorithm to prevent round-off errors in case the step size is non-integral.

Example:

```
> nseq(<< x, y -> x:x^2 + y >>, 1, 5, 1, 10):
seq(1:11, 2:14, 3:19, 4:26, 5:35)
```

See also: **map**.

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: **values**.

print (··· [, option])

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'`:<any string> is given as the last argument, then **print** separates multiple values with the given <string>, otherwise `'\t'` is used. If the option `'nonewline':true` is passed, then Agena does not print a final newline when finishing output. Note that these two options cannot be used together.

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 `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**.

`purge (obj [, pos])`

Removes from table or sequence `obj` 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 or sequence, so that a call `purge(obj)` removes the last element of `obj`.

Use the **delete** *element* **from** *table* statement if you want to remove any occurrence of the table value *element* from a table or sequence.

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.

See also: **put**.

`put (obj, [pos,] value)`

Inserts element `value` at position `pos` in table or sequence `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 table or sequence, so that a call `put(obj, value)` inserts `value` at the end of `obj`.

Use the **insert** *element* **into** *table* statement if you want to add an element at the current end of a table, for it is much faster.

The function returns nothing.

See also: **purge**.

`qsadd (obj)`

Raises all numeric values in table or sequence `obj` to the power of 2 and sums up these powers. The return is a number. If `obj` is empty or consists entirely of non-numbers, **null** is returned. If the table or sequence contains numbers and other objects, only the powers of the numbers are added. Entries with non-numeric keys are ignored.

See also: **sadd**.

`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, sequence, or pair, `index` any value different from **null**, and `value` any 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, or sequence.

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 the Agena language and included in the `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 with 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.

See also: **run**, **with**.

recurse (obj, f)

Checks each element of the structure `obj` (a table, set, pair, or sequence) by applying a function `f` on each of its elements. `f` must be a function of one argument and return either **true** or **false**.

With tables, all the entries and keys are scanned.

With sequences, only the entries (not the keys) are scanned.

The function performs a recursive descent if it detects tables, sets, pairs, or sequences in `obj` so that it can find elements in deeply nested structures.

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: **has**.

_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**.

remove (f, obj [, ··· [, newarray=true]])

Returns all values in table, set, or sequence `obj` that do not satisfy a condition determined by function `f`, as a new table, set, or sequence. 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/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.

```
> 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 is 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]
```

See also: **countitems**, **map**, **select**, **selectremove**, **subs**, **unique**, **zip**.

restart

Restarts an Agena session. No argument is needed.

During start-up, Agena stores all initial values, e.g. package tables assigned, in a global variable called **_origG**. Tables are copied, too, so their contents cannot be altered in a session.

If the Agena session is restarted with **restart**, all values in the Agena environment are unassigned including the environment variable **_G**, but except of **_origG**, mainlibname, and **libname** (**mainlibname** and **libname** are reset to their original values if the kernel setting `environ.kernel('libnamereset')` results to **true**, however.) Then all entries in **_origG** are read and assigned to the new environment.

After this, the library base file `agena.lib` and thereafter the initialisation file `agena.ini` - if present - are read and executed. Finally, restart runs a garbage collection.

The return of the function is **false** if evaluation of **_origG** failed because it is no longer a table (which should never happen). Otherwise, the return is **true**.

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**.

sadd (obj)

Sums up all numeric values in table or sequence `obj`. The return is a number. If `obj` is empty or consists entirely of non-numbers, **null** is returned. If the object contains numbers and other objects, only the numbers are added. Entries with non-numeric keys are ignored.

See also: **qsadd**, **calc.fsum**, **stats.sum**.

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 the Agena language and included in the `library.agn` file.

See also: **read**, **io.writefile**.

select (f, obj [, ··· [, newarray=true]])

Returns all values in table, set, or sequence `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.

```
> 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 is 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]
```

See also: **countitems**, **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 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 is 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`$| \in [1 .. 31]$.

Please note that if `x` is negative, then the result is **sign**(`x`) * **setbit**(**abs**(`x`), `pos`, `bit`), thus abstracting from the internal hardware representation of `x`.

The function is written in the Agena language and included in the `library.agn` file.

See also **getbit**.

`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**.

`settype (obj [, ···], str)`
`settype (obj [, ···], null)`

In the first form the function sets the type of one or more procedures, sequences, tables, sets, or pairs `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.

See also: **gettype**.

`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 strings, the operator returns the number of characters in string `obj`, i.e. the length of `obj`.

See also: **environ.attrib**, **strings.utf8size**.

`sort (obj [, f])`

Sorts table or sequence elements in a given order, in-place, from `obj[1]` to `obj[n]`, where n 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).

See also: **sorted**, **stats.issorted**, **skycrane.sorted**, **stats.sorted**.

Example:

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

> sort(s, << x, y -> x > y >>)

> s:
[3, 2, 1]

> s := seq(1:'a', 1.1:'b', 1.2:'c');

> sort(s, << x, y -> left(x) > left(y) >>)

> s:
seq(1.2:c, 1.1:b, 1:a)
```

sorted (obj [, f])

Sorts table or sequence elements in `obj` in a given order, but - unlike sort - not in-place, and non-destructively. 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)

Substitutes all occurrences of the value `x` in the table, set, or sequence `obj` with the value `v`. More than one substitution pair can be given. The substitutions are performed sequentially and 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.

See also: **countitems**, **map**, **remove**, **select**, **zip**.

`time ()`

Returns the time till start-up in seconds as a number.

Calling **time** only once does not necessarily return a real amount of time; instead conduct a subtraction by calling **time** again to get correct results.

See also: **os.difftime**, **os.time**.

`top (obj)`

With the table array or sequence `obj`, the operator returns the element with the largest index. If `obj` is empty, it returns **null**.

See also: **bottom**.

`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.

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: **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 or sequence, 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: **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 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: **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'`, `'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: **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 special 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 is returned, i.e. `'null'` (the string, not the value **null**), `'number'`, `'string'`, `'boolean'`, `'table'`, `'set'`, `'sequence'`, `'pair'`, `'complex'`, `'procedure'`, `'thread'`, and `'userdata'`.

See also: **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 **unique** operator removes all holes (`missing keys`) and removes multiple occurrences of the same value, if present. The return is a new table with the original table unchanged.

With a sequence `obj`, the **unique** operator removes multiple occurrences of the same value, if present. The return is a new sequence with the original sequence unchanged.

See also: **tables.entries**.

unpack (obj, [, i [, j]])

Returns the elements from the given table or sequence `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 the argument list of a call to a function or operator, in most cases only the first return of **unpack** is propagated to the function or operator. However, the behaviour is not arbitrary.

See also: **ops**, **values**.

values (obj, i_1 [, i_2, ···]])

Returns the elements i_k from the given table or sequence `obj`. This operator is equivalent to

```
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: **ops**, **select**, **unpack**.

whereis (obj, x)

Returns the indices for a given value `x` in table or sequence `obj` as a new table or sequence, respectively.

See also: **tables.indices**.

`with (packagename [, key1, key2, ···])`

Assigns short names to package procedures such that:

```
name := packagename.name
```

The function works as follows:

- In both forms, **with** 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 is returned.

- Next, **with** 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 with 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, **with** 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 **with** 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 **with**, see **readlib**.

  Note that **with** executes any statements (and thus also any assignment) included in the file `packagename`.agn.

The function is written in the Agena language and included in the `library.agn` file.

See also: **readlib**, **run**, **register**.

`write ([fh,] v₁ [, v₂, ···] [, delim = <str>])`

This function prints one or more numbers or strings v_k 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 or strings v_k 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)`

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. 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 [, ···])`

This function zips together either two sequences 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 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`.

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**.

7.2 String Manipulation

Summary of Functions:

Search

atendof, in, instr, strings.find, strings.glob, strings.match, strings.mfind.

Insertion, Substitution, and Deletion

replace, strings.gsub, strings.include, strings.remove.

Extraction

split, strings.fields, strings.gmatch, strings.gmatches, strings.separate.

Queries

abs, strings.dleven, strings.isabbrev, strings.isalpha, strings.isalphanumeric, string.isalphaspace, string.isalphaspec, strings.isblank, strings.iscenumeric, strings.isending, strings.isfloat, 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.isspace, strings.isspec, strings.isupperalpha, strings.isupperlatin, strings.isutf8.

Counting

size, strings.hits, 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.

Conversion

&, join, tonumber, tostring, strings.diamap, strings.reverse, strings.tolatin, strings.toutf8, strings.transform.

Manipulation

map, strings.repeat, strings.tobytes, strings.tochars.

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					

7.2.1 Kernel Operators and Basic Library Functions

s1 & s2

This binary operator concatenates two strings `s1`, `s2` and returns a new string. `s1` or `s2` may also be a number. In this case the number is converted to a string and then concatenated with the other operand.

See also: **join**.

s1 atendof s2

This binary operator checks whether a string `s2` ends in a substring `s1`. If true, the position of the position of `s1` in `s2` is returned; otherwise **null** is 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.isabbrev**, **strings.isending**.

s1 in s2

This binary operator checks whether the 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**, **strings.isabbrev**, **strings.isending**.

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 is returned.

See also: **strings.fields**, **strings.separate**.

abs (s)

With strings, the operator returns the numeric ASCII value of the given character `s` (a string of length 1).

instr (s, pattern [, init] [, plain] [, 'reverse'] [, 'borders'])

Looks for the first match of `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 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` is 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 7.2.3. **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 **string.find** instead which is twice as fast.

See also: **atendof**, **in**, **strings.isabbrev**, **strings.isending**, **strings find**.

join (obj [, sep [, i [, j]]])

Concatenates all string values in the table or sequence `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)

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 above mentioned diacritics). The operator leaves all other characters unchanged.

See also: **strings.isolower**, **upper**.

map (f, s [, ···])

This operator 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`.

replace (s1, s2, s3)

replace (s1, obj)

replace (s1, pos, s2)

In the first form, the operator replaces all occurrences of string `s2` in string `s1` by string `s3`.

In the second form, the operator receives a string `s1` and a table or sequence `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 operator 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 operator does not support pattern matching, use **strings.gsub** instead.

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** is returned.

`tostring (e)`

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.

`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.rtrim**.

`upper (s)`

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 above mentioned diacritics). The operator leaves all other characters unchanged.

See also: **lower**, **strings.capitalise**, **strings.isoupper**.

7.2.2 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.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.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.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.

strings.diamap (s)

The function corrects problems in the Solaris, Linux, OS/2, Windows, and DOS consoles with diacritics and ligatures read in from a text file (even .agn programme files) by mapping them to their correct character codes. It takes a strings `s`, applies the mapping, and returns a new string. All other characters are returned unchanged.

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 codepage 1252. If you want to use another codepage, edit the `_c2f` and `_f2c` tables in the `library.agn` file accordingly.

strings.dump (f)

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.

strings.fields (s, i_1 [, i_2, ···] [, delim])

Extracts the given fields (columns) in string `s`. The field position i_1, i_2, etc. are integers. The field positions may be negative, denoting fields counted from the right end of `s`.

An optional string `delim` may be passed as the last argument to denote the character or character sequence that separates the individual fields. The default for `delim` is the white space.

The return is a sequence of the fields (strings).

See also: **split**, especially if you want to retrieve all fields in a string.

strings.find (s, pattern [, init [, plain]])

Looks for the first match of `pattern` in the 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).

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 7.2.3), 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** operator, **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 follows the same rules as the `printf` family of standard C functions. The only differences are that the options/modifiers *, l, L, n, p, and h are not supported and that there is an extra option, q. The q option formats a string in a form suitable to be safely read back by the Agena interpreter: the string is written between double quotes, and all double quotes, newlines, embedded zeros, and backslashes in the string are correctly escaped when written. 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 options c, d, E, e, f, g, G, i, o, u, X, and x all expect a number as argument, whereas q and s expect a string.

This function does not accept string values containing embedded zeros.

strings.glob (s, pattern)

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.

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 7.2.3.

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 the Agena language and included in the `library.agn` file.

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.

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 7.2.3) 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 is returned.

See also: **strings.remove**.

strings.isabbrev (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 7.2.3) 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, 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.isalpha (s)

Checks whether the string s consists entirely of alphabetic letters (including diacritics) and returns **true** or **false**.

See also: **strings.isisoalpha** , **strings.islatin**.

strings.isalphanumeric (s)

Checks whether the string s consists entirely of numbers or alphabetic letters (including diacritics) and returns **true** or **false**.

See also: **strings.islatinnumeric**.

strings.isalphaspace (s)

Checks whether the string s consists entirely of alphabetic letters (including diacritics) and/or a white space and returns **true** or **false**.

strings.isalphaspec (s)

Checks whether the string s consists entirely of the Latin letters a to z, A to Z, or the following special characters:

white space `¿ ? ¡ ! " # $ @ § % & ' ` * / + - . , ; ( ) [ ] { } | ¦ \ ^ _ ~ = < >`

and returns **true** or **false**.

See also: **strings.isspec** , **strings.isalphaspace** .

strings.isblank (s)

Checks whether the string `s` consists entirely white spaces or tabulators (\t) and returns **true** or **false**.

See also: **strings.isisospace** , **strings.isspace** .

strings.iscenumeric (s)

Checks whether the string `s` consists entirely of the digits 0 to 9 or digits 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.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 7.2.3) 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, 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.isabbrev**, **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.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)

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**.

See also: **strings.isalpha**.

strings.islatinnumeric (s)

Checks whether the string s consists entirely of numbers or Latin letters 'a' to 'z' and 'A' to 'Z', and returns **true** or **false**.

See also: **strings.isalphanumeric**.

strings.isloweralpha (s)

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**.

See also: **strings.isisolower**, **strings.isupperalpha**.

strings.islowerlatin (s)

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**.

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.

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.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 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 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.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 the following special characters:

white space `¿ ? ¡ ! " # $ @ § % & ' ` * / + - . , ; ( ) [ ] { } | ¦ \ ^ _ ~ = < >`

and returns **true** or **false**.

See also: **strings.isalphaspec**, **strings.isspace**, **strings.ismagic**.

`strings.isupperalpha (s)`

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**.

See also: **strings.isisoupper**, **strings.isloweralpha**.

`strings.isupperlatin (s)`

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**.

See also: **strings.islowerlatin**.

strings.isutf8 (s)

Detects that 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.

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.isisoalpha** .

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`, 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.

It does not remove spaces or the given character within the `actual` part of the string.

See also: **trim** operator, **strings.ltrim** , **strings.rtrim**.

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.

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 is 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, see Chapter 4.7.8.

See also: **strings.gmatch**.

`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 7.2.3), 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** operator, **strings.find**, **strings.mfind**.

`strings.remove (s, pos [, len])`

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.

See also: **replace**, **strings.include**.

`strings.repeat (s, n)`

Returns a string that is the concatenation of `n` copies of the string `s`.

`strings.reverse (s)`

Returns a string that is the string `s` 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`, 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.

See also: **trim** operator, **strings.lrtrim**, **strings.ltrim**.

`strings.separate (s, d)`

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 includes one or more characters given in `d`, or if `s` or `d` are empty strings, the function returns **fail**.

```
> a := strings.separate('a word, another word.', ' .,'):
seq(a, word, another, word)
```

See also: **split** operator.

`strings.tobytes (s)`

Converts a string `s` into a sequence of its numeric ASCII codes. If the string is empty, an empty sequence is returned.

Note that numerical codes are not necessarily portable across platforms.

`strings.tochars (···)`

Receives zero or more integers 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.

Note that numerical codes are not necessarily portable across platforms.

`strings.tolatin (s)`

Creates a dynamically allocated copy of string `s`, changing the encoding from UTF-8 to ISO-8859-15. Unsupported code points are ignored. The return is a string. ISO-8859-15 is ISO-8859-1 plus the Euro symbol.

See also: **strings.toutf8**.

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: 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 is issued.

Note that numerical codes are not necessarily portable across platforms.

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.

7.2.3 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 with 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.

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.

7.3 Table Manipulation

Summary of Functions:

Queries

> countitems, filled, in, size, tables.maxn.

Retrieving Values

> getentry, unique, unpack, values, tables.entries, tables.indices.

Operations

> copy, map, qsadd, sadd, remove, select, selectremove, sort, sorted, subs, zip.

Relational Operators

> =, ==, <>.

Cantor Operations

> intersect, minus, subset, union, xsubset.

Assignment

> dimension, tables.allocate.

7.3.1 Kernel Operators

Most of the following functions have been built into the kernel as unary operators, with the exception of **map** and **zip**.

`copy (t)`

The operator copies the entire contents of a table `t` into a new table. See Chapter 7.1 for more information.

`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**.

`dimension (a:b [, c:d] [, init])`

Creates a table of dimension 1 or 2 with arbitrary index ranges and an optional default for all its elements. See Chapter 7.1 for more information.

`filled (t)`

Checks whether table `t` contains at least one element. The return is **true** or **false**. The operator works with dictionaries, as well.

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 [, ···])`

Maps the function `f` on all elements of a table `t`. See **map** in Chapter 7.1 for more information. See also: **countitems**, **remove**, **select**, selectremove**, subs**, and **zip**.

`qsadd (t)`

Raises all numeric values in table `t` to the power of 2 and sums up these powers. See **qsadd** in Chapter 7.1 for more information. See also: **sadd**.

`remove (f, t [, ··· [, newarray=true]])`

Returns all values in table `t` that do not satisfy a condition determined by function `f`. See **remove** in Chapter 7.1 for more information. See also: **map**, **select**, **selectremove**, **subs**, **zip**.

sadd (t)

Sums up all numeric values in table t. See **sadd** in Chapter 7.1 for more information. See also: **qsadd**.

select (f, t [, ··· [, newarray=true]])

Returns all values in table t that satisfy a condition determined by function f. See **select** in Chapter 7.1 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 7.1 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. See also: **environ.attrib**.

sort (t [, comp])

Sorts table t in a given order, and in-place. See **sort** in Chapter 7.1 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 7.1 for more information. See also: **sort**, **skycrane.sorted**, **stats.issorted**, **stats.sorted**.

subs (x:v [, ···], t)

Substitutes all occurrences of value x in table t with value v. See **subs** in Chapter 7.1 for more information. See also: **map**, **remove**, **select**, **zip**.

unique (t)

The **unique** operator removes all holes (`missing keys`) in a table t and removes multiple occurrences of the same value, if present. See **unique** in Chapter 7.1 for more information.

values (t, i_1 [, i_2, ···]])

Returns the elements from the given table t in a new table. This operator 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 7.1 for more information. See also: **map**, **remove**, **select**, **subs**, **zip**.

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.

`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.

`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.

`c in t`

Checks whether the table `t` contains the value `c` and returns **true** or **false**. The search is linear.

t1 <u>intersect</u> 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.

t1 <u>minus</u> 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.

t1 <u>subset</u> 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 <u>union</u> t2

Concatenates two tables `t1` and `t2` simply by copying all its elements - even if they occur multiple times - to a new table.

t1 <u>xsubset</u> 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 7.1 Basic Functions.

7.3.2 tables Library

This library provides generic functions for table, and also sequence 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, $value_2$, ···, 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.

`tables.entries (t)`

Returns all entries of table `t` (not its keys) in a new table array.

See also: **tables.indices** , **unique**, **whereis**.

`tables.indices (t)`

Returns all keys of table `t` in an unsorted new table.

See also: **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.)

7.4 Set Manipulation

Summary of Functions:

Queries

filled, **in**, **size**.

Retrieving Values

unpack.

Operations

copy, **map**, **remove**, **select**, **selectremove**.

Relational Operators

=, ==, <>.

Cantor Operations

intersect, **minus**, **subset**, **union**, **xsubset**.

The following functions have been built into the kernel as unary operators.

`copy (s)`

The operator copies the entire contents of a set `s` into a new set. See Chapter 7.1 for more information.

`filled (s)`

The operator checks whether a set `s` contains at least one element. The return is **true** or **false**.

`map (f, s [, ···])`

Maps the function `f` on all elements of a set `s`. See **map** in Chapter 7.1 for more information. See also: **countitems**, **remove**, **select**, selectremove, **subs**, and **zip**.

`remove (f, s [, ··· [, newarray=true]])`

Returns all values in set `s` that do not satisfy a condition determined by function `f`. See **remove** in Chapter 7.1 for more information. See also: **map**, **select**, **selectremove**, **subs**, **zip**.

select (f, s [, ··· [, newarray=true]])

Returns all values in set `s` that satisfy a condition determined by function `f`. See **select** in Chapter 7.1 for more information. See also: **map**, **remove**, **selectremove**, **subs**, **zip**.

selectremove (f, s [, ··· [, newarray=true]])

Returns all values in set `s` that satisfy and do not satisfy a condition determined by function `f`, in two tables. See **selectremove** in Chapter 7.1 for more information. See also: **map**, **remove**, **select**, **subs**, **zip**.

size (s)

Returns the number of items in a 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 like the = operator.

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.

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.

s1 <u>minus</u> s2

Searches all items in set `s1` that are not items in set `s2` and returns them as a set. The search is linear.

s1 <u>subset</u> 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 <u>union</u> s2

Concatenates two sets `s1` and `s2` simply by copying all its items to a new set.

s1 <u>xsubset</u> 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.

7.5 Sequence Manipulation

Summary of Functions:

Queries

> **countitems**, **filled**, **in**, **size**.

Retrieving Values

> **getentry**, **unique**, **unpack**, **values**.

Operations

> **copy**, **map**, **qsadd**, **remove**, **select**, **selectremove**, **sadd**, **sort**, **sorted**, **subs**, **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.

`copy (s)`

The operator copies the entire contents of a sequence `s` into a new sequence. See Chapter 7.1 for more information.

`countitems (item, s)`

`countitems (f, s [, ···])`

Counts the number of occurrences of an `item` in the structure (table, set, or sequence) `s`. For further information, see Chapter 7.1.

`filled (s)`

The operator checks whether the sequence `s` contains at least one element. The return is **true** or **false**.

getentry (s [, k_1, ···, k_n])

Returns the entry `s[`k_1`, ···, `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 [, ···])

Maps the function `f` on all elements of a sequence `s`. See **map** in Chapter 7.1 for more information. See also: **remove**, **select**, **subs**, **zip**.

qsadd (s)

Raises all numeric values in sequence `s` to the power of 2 and sums up these powers. See **qsadd** in Chapter 7.1 for more information. See also: **sadd**.

remove (f, s [, ··· [, newarray=true]])

Returns all values in sequence `s` that do not satisfy a condition determined by function `f`. See **remove** in Chapter 7.1 for more information. See also: **map**, **select**, **subs**, **zip**.

sadd (s)

Sums up all numeric values in sequence `s`. See **sadd** in Chapter 7.1 for more information. See also: **qsadd**.

select (f, s [, ···])

Returns all values in sequence `s` that satisfy a condition determined by function `f`. See **select** in Chapter 7.1 for more information. See also: **map**, **remove**, **subs**, **zip**.

selectremove (f, s [, ··· [, newarray=true]])

Returns all values in sequence `s` that satisfy and do not satisfy a condition determined by function `f`, in two tables. See **selectremove** in Chapter 7.1 for more information. See also: **map**, **remove**, **select**, **subs**, **zip**.

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 7.1 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 7.1 for more information. See also: **sort**, **skycrane.sorted**, **stats.issorted**, **stats.sorted**.

subs (x:v [, ···], s)

Substitutes all occurrences of the value `x` in sequence `s` with the value `v`. See **subs** in Chapter 7.1 for more information. See also: **map**, **remove**, **select**, **zip**.

unique (s)

With a sequence `s`, the **unique** operator removes multiple occurrences of the same item, if present in `s`. See **unique** in Chapter 7.1 for more information.

values (s, i_1 [, i_2, ···]])

Returns the elements from the given sequence `s` in a new sequence. This operator 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 7.1 for more information. See also: **map**, **remove**, **select**, **subs**.

See also: **bintersect**, **bisequal**, **bminus**, **purge**, **put** in Chapter 7.1 Basic Functions.

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. `seq(1, 1) = seq(1)` → true, `seq(1, 2) xsubset seq(1, 1, 2, 2, 3, 3)` → true.

s1 <u>=</u> 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 <u>==</u> 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 <u><></u> 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 <u>in</u> s

Checks whether the sequence `s` contains the value `c` and returns **true** or **false**. The search is linear.

s1 <u>intersect</u> s2

Searches all values in sequence `s1` that are also values in sequence `s2` and returns them in a sequence. The search is quadratic.

s1 <u>minus</u> s2

Searches all values in sequence `s1` that are not values in sequence `s2` and returns them as a sequence. The search is quadratic.

s1 <u>subset</u> 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 <u>union</u> s2

Concatenates two sequences `s1` and `s2` simply by copying all its elements - even if they occur multiple times - to a new sequence.

`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.

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.

7.6 llist - Linked Lists

As a *plus* package, the **llist** package is not part of the standard distribution and must be activated with the **readlib** or **with** functions, e.g. `readlib 'llist'`.

7.6.1 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.[22]" 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 at least six times faster even when conducting a read operation with each write operation.

This package implements linked lists using a table and pairs. The master table of the user-defined type `'llist'` contains the current number of elements in the list in its `'sizeof'` field, a link to the first element (i.e. its node) in its `'nodes'` field, and a link to the last element (i.e. its node) in its `'last'` field. Each node is represented by a pair of the user-defined type `'node'`, where the value is kept in its left-hand side, and the reference to the node containing the next element in its right-hand side.

Thus, the structure `llist(-1, 0)` created by the statements

```
> L := llist.list():
llist(next ~ node(), sizeof ~ 0)

> llist.append(L, 0); llist.prepend(L, -1);
```

actually looks as follows:

```
llist(
   last ~ node(0, null),
   next ~ node(-1, node(0, null)),
   sizeof ~ 2)
```

Metamethods exist to support printing, indexing, and indexed assignments; the **size** and **in** operators are also supported.

---

[22] 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.

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.

For an example of how to use linked lists, see Chapter 6.25.

7.6.2 Functions

`llist.append (l, obj [, ···])`

Appends one or more elements `obj` which may be of any type, to the linked list `l`, in sequential order. There is no return.

The function is written in the Agena language and included in the `llist.agn` file.

See also: **llist.prepend**, **llist.put**.

`llist.iterate (l [, n])`

Returns an iterator function that when called returns the next value in the 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 elements after index `n`.

The function is written in the Agena language and included in the `llist.agn` file.

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 < L.sizeof do
>    inc c;
>    print(f())
> od;
1
null
2
```

`llist.list ([···])`

The function creates a new linked list and optionally stores all of the given elements in it. The return is a table of user-type `'llist'`.

The function is written in the Agena language and included in the `llist.agn` file.

`llist.listtotable (l)`

The function creates a new table and copies all elements in the linked list `l` into it, in sequential order. The return is the table. If there are no elements in `l`, an empty table is returned. If the list includes **null**s, the resulting table will contain holes.

The function is written in the Agena language and included in the `llist.agn` file.

`llist.prepend (l, obj)`

Prepends an element `obj`, which may be of any type, to the linked list `l`. There is no return.

The function is written in the Agena language and included in the `llist.agn` file.

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 nothing, but issues an error if there is no element (i.e. node) at index `n`.

The function is written in the Agena language and included in the `llist.agn` file.

`llist.put (l, n, obj)`

The function inserts the given element `obj` at position `n` into linked list `l`. The original element at position `n` is not deleted - it and all of its successors are `shifted` to open space. The function returns nothing, or issues an error if the index is out-of-range.

The function is written in the Agena language and included in the **llist.agn** file.

See also: **llist.append**, **llist.prepend**.

7.7 bags - Mulitsets

As a *plus* package, the **bags** package is not part of the standard distribution and must be activated with the **readlib** or **with** functions, e.g. `readlib 'bags'`.

7.7.1 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:

```
> readlib 'bags';

> orders := seq(
>   'Programming in Lua', 'Moon Lander', 'Einmischungen', 'Unser Schmidt',
>   'Programming in Lua', 'Moon Lander', 'Einmischungen', 'Unser Schmidt');

> books := bags.bag(unpack(orders));

> books.Einmischungen:
2
```

For a further order, just enter

```
> bags.include(books, 'Agena')

> books:
bag('Agena' ~ 1, Programming in Lua ~ 2, Unser Schmidt ~ 2, Einmischungen ~
2, Moon Lander ~ 2)
```

A customer has cancelled his previous order:

```
> bags.remove(books, 'Agena'):

> books:
bag(Programming in Lua ~ 2, Unser Schmidt ~ 2, Einmischungen ~ 2, Moon
Lander ~ 2)
```

7.7.2 Functions

`bags.attribs (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.

`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.include (b, obj [, ···])`

The function inserts all of the given elements `obj`, etc. into bag `b`.

The function returns nothing.

`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 **+** (for union), **-** for difference set, ***** for intersection, and **in** for searching an object.

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
Unser Schmidt   2
Einmischungen   2
Moon Lander     2
```

7.8 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 in the Agena language). While functions can be overwritten with self-defined versions, operators cannot be overwritten.

Summary of Functions:

Basic Arithmetic Operators

+, -, *, /, /*, fma.

Integer Division

\, %, irem, iqr, modf.

Exponentiation

^, **, exp, expx2, frexp, ldexp.

Roots

hypot, root, sqrt.

Logarithms

ln, log, log2, log10.

Trigonometric Functions

cos, cot, csc, sec, sin, tan.

Inverse Trigonometric Functions

arccos, arccsc, arccot, arcsec, arcsin, arctan, arctan2.

Hyperbolic Functions

cosh, coth, csch, sech, sinh, tanh.

Inverse Hyperbolic Functions

arccosh, arccsch, arccoth, arcsech, arcsinh, arctanh.

Miscellaneous

abs, erf, erfc, even, heaviside, sign, math.gcd, math.lcm, math.max, math.min.

Miscellaneous Complex Functions

argument, conjugate, polar.

Gamma, etc.

beta, binomial, fact, gamma, lngamma.

Bessel Functions

besselj, bessely.

Rounding Functions

ceil, entier, int, roundf.

Relational Operators

=, ==, <, >, <=, =>, <>, approx.

Numbers

finite, float, frac, gethigh, getlow, isint, isnegative, isnegint, isnonneg, isnonnegint, isnonposint, isnumber, isnumeric, isposint, ispositive, sethigh, setlow, math.fraction, math.ndigits, math.nextafter.

Random Numbers

math.random, math.randomseed.

Bases and Conversion

math.convertbase, math.norm, math.todecimal, math.toradians.

Primes

math.isprime, math.nextprime, math.prevprime.

Bitwise Operators

&&, ~~, ||, ^^, getbit, setbit, shift.

7.8.1 Operators and Basic Functions

x + y

The operator adds two numbers; returns a number.

x - y

The operator subtracts two numbers; returns a number.

x * y

The operator multiplies two numbers; returns a number.

x / y

The operator divides two numbers; returns a number.

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 adds the given percentage y to x.

x -% y

The operator subtracts the given percentage y from x.

x \ y

The operator performs an integer division of two numbers, and returns a number. The integer division is defined as: $x \backslash y = \mathrm{sign}(x) * \mathrm{sign}(y) * \mathrm{entier}(|\frac{x}{y}|)$.

x % y

The modulus operator conducts the operation $x \% y = x - \mathrm{entier}(\frac{x}{y})*y$.

x ^ y

The operator performs an exponentiation of real or complex x with a rational power y.

x ** y

The operator exponentiates the real or complex number x with the integer power y. This operator is at least 50 % faster than the ^ operator.

x && y

Bitwise `and` operation on two numbers x and y. By default, the operator internally calculates with signed integers. You can change this to unsigned integers by using the **kernel** function with the **signedbits** option. See also: **environ.kernel** in Chapter 7.20.

~~ (x)

Bitwise complementary operation on the number x. By default, the operator internally calculates with signed integers. You can change this to unsigned integers by using the **environ.kernel** function with the **signedbits** option. See also: **environ.kernel** in Chapter 7.20.

x || y

Bitwise `or` operation on two numbers x and y. By default, the operator internally calculates with signed integers. You can change this to unsigned integers by using the **environ.kernel** function with the **signedbits** option. See also: **environ.kernel** in Chapter 7.20.

x ^^ y

Bitwise `exclusive-or` operation on two numbers x and y. By default, the operator internally calculates with signed integers. You can change this to unsigned integers by using the **environ.kernel** function with the **signedbits** option. See also: **environ.kernel** in Chapter 7.20.

x shift y

Bitwise shift operation. If the right-hand side y is a positive integer, the bits in x are shifted to the left (multiplication with 2), else they are shifted to the right (division by 2). By default, the operator internally calculates with signed integers. You can change this to unsigned integers by using the **environ.kernel** function with the **signedbits** option. See also: **environ.kernel**.

abs (x)

If x is a number, the **abs** operator returns the absolute value of x. Complex numbers are supported.

See also: **argument**, **polar**.

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.

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 the Agena language and included in the library.agn file. The function works on both numbers and complex values.

arccsc (x)

Returns the inverse cosecant of x (in radians). The function works on both numbers and complex values. The function is implemented in the Agena language and included in the 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 the Agena language and included in the 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 the Agena language and included in the 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 function works on both numbers and complex values. The function is implemented in the Agena language and included in the `library.agn` file.

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 the Agena language and included in the `library.agn` file.

arcsin (x)

Computes the inverse sine operator (`x` in radians). Complex numbers are supported.

arcsinh (x)

Returns the inverse hyperbolic sine of `x` (in radians). The function is implemented in the Agena language and included in the `library.agn` file. The function works on both numbers and complex values.

arctan (x)

Computes the inverse tangent operator (`x` 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 the Agena language and included in the `library.agn` file.

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**, **polar**.

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. The function returns **undefined**, if n or k are negative, or if at least one of its arguments is not an integer.

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.

ceil (x)

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. The function is implemented in the Agena language and included in the `library.agn` file.

See also: **entier**, **int**, **roundf**.

conjugate (z)

The conjugate x-I*y of the complex value z=x+I*y. If z is of type number, it is simply returned.

cos (x)

The operator returns the cosine of x (x in radians). Complex numbers are supported.

cosh (x)

The operator returns the hyperbolic cosine of x. Complex numbers are supported.

cot (x)

Returns the cotangent -tan($\frac{\pi}{2}$+x) as a number. The function is implemented in the Agena language and included in the `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. The function is implemented in the Agena language and included in the `library.agn` file. The function works on both numbers and complex values.

csc (x)

Returns the cosecant $\frac{1}{\sin(x)}$ as a number. The function is implemented in the Agena language and included in the `library.agn` file. The function works on both numbers and complex values.

csch (x)

Returns the hyperbolic cosecant as a number. The function is implemented in the Agena language and included in the `library.agn` file. The function works on both numbers and complex values.

entier (x)

The operator rounds `x` downwards to the nearest integer. Complex numbers are supported.

See also: **ceil**, **int**, **roundf**.

erf (x)

Returns the error function of `x`. It is defined by $\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_{t=0}^{x} e^{-t \wedge 2}$. The function works on both numbers and complex values.

See also: **erfc**.

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**.

even (x)

Checks whether `x` is even. The operator returns **true** if `x` is even, and **false** otherwise. With the complex value `x`, the operator returns **fail**.

exp (x)

Exponential function; the operator returns the value e^x. Complex numbers are supported.

expx2 (x, sign)

Computes either e^{x^2} if `sign` ≥ 0, or e^{-x^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.

fact (n)

Returns the factorial of `n`, i.e. the product of the values from 1 to `n`. If `n` is not an integer or if `n` is negative, the function returns **undefined**. The function is implemented in the Agena language and included in the `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).

finite (x)

Checks whether `x` is a number and neither ±**infinity** nor **undefined** (NaN). The operator returns **true** or **false**. If `x` is a complex number, it returns **fail**.

float (x)

Checks whether the number `x` is a float, i.e. not an integer, and returns **true** or **false**. If `x` is not a number, the operator returns **fail**.

fma (x, y, z)

Performs the floating-point 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.

frac (x)

Returns the fractional part of the number `x`, i.e. `x - int(x)`. The function is implemented in the Agena language and included in the `library.agn` file.

See also: **modf**.

frexp (x)

Returns `m` and `e` such that $x = m2^e$, `e` is an integer and the absolute value of `m` is in the range [0.5, 1) (or zero when `x` is zero).

See also: **ldexp**.

gamma (x)

The gamma function Γ `x`. `x` may be a number or complex value.

See also: **lngamma**.

gethigh (x)

Returns the higher bytes of a number x as an integer. This operator does not support complex numbers. See also: **getlow**, **sethigh**.

getlow (x)

Returns the lower bytes of a number x as an integer. This operator does not support complex numbers. See also: **gethigh**, **setlow**.

heaviside (x)

The Heaviside function. Returns 0 if $x < 0$, **undefined** if $x = 0$, and 1 if $x > 0$. The function is implemented in the Agena language and included in the `library.agn` file.

hypot (x, y)

Returns $\sqrt{x^2+y^2}$ with x, y numbers. This 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**. The return is a number.

See also: **root**, **sqrt**.

int (x)

Rounds x to the nearest integer towards zero. The operator also supports complex numbers.

See also: **ceil**, **entier**, **roundf**.

iqr (x, y)

Computes both the integer quotient and the integer remainder 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, irem(x, y)
   fi
end;
```

irem (x, y)

Evaluates the remainder of an integer division x/y (with x, y two Agena numbers). The return is a number. The remainder r has the same sign as the numerator. If x and y are integers and q the integer quotient of x and y, then the function returns the remainder such that $x = y*q + r$, $|r| < |y|$ and $x*r \geq 0$.

See also: \, **%**.

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**.

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**, **innonneg**, **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**.

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: **notisposint**.

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 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: **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**.

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: ln, log, log10.

log10 (x)

Returns the base-10 logarithm of the number or complex number x.

See also: ln, log, log2.

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.

See also: \, %, /%, frac, int.

polar (z)

Transforms the complex number z in Cartesian notation or the number z to polar form. 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.

root (x, n)

Returns the non-principal n-th root of the number or complex value x. n must be an integer.

See also: hypot, sqrt.

roundf (x [, d])

Rounds the number x to its d-th digit. Return is a number. If d is omitted, the number is rounded to the nearest integer. The following Agena code explains the algorithm used:

```
roundf := proc(x, digs) is
   local d;
   if digs = null then d := 0 else d := digs fi;
   return int((10^d)*x + sign(x)*0.5) * (10^(-d))
end;
```

See also: ceil, entier, int.

sec(x)

Returns the secant $\frac{1}{\cos(x)}$ as a number. The function is implemented in the Agena language and included in the `library.agn` file. The function works on both numbers and complex values.

sech(x)

Returns the hyperbolic secant as a number. The function is implemented in the Agena language and included in the `library.agn` file. The function works on both numbers and complex values.

sethigh (x, i)

Sets the higher bytes of the number `x` to the integer `i`, and returns the new number. This operator does not support complex numbers. See also: **setlow**, **gethigh**.

setlow (x, i)

Sets the lower bytes of the number `x` to the integer `i`, and returns the new number. This operator does not support complex numbers. See also: **sethigh**, **getlow**.

sign (x)

Determines the sign of the number or complex value `x`. If `x` is a complex value, 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.

If `x` is **undefined**, **sign** returns **undefined**.

sin (x)

The operator returns the sine of `x` (`x` in radians). Complex numbers are supported.

sinh (x)

The operator returns the hyperbolic sine of `x`. Complex numbers are supported.

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: **hypot**, **root**.

tan (x)

The operator returns the tangent of `x` (`x` in radians). Complex numbers are supported.

tanh (x)

The operator returns the hyperbolic tangent of `x` (`x` in radians). Complex numbers are supported.

7.8.2 math Library

This library is an interface to the standard C math library. It provides all miscellaneous functions inside the table `math`.

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 the Agena language and included in the `library.agn` file.

math.fraction (x [, err])

Given a number `x`, this function outputs two integers, the numerator n and the denominator d, such that `x` := n / d to an accuracy epsilon := | (x - n/d) / x | ≤ `err`. The error `err` should be a non-negative number, and by default is 0.

The returns are three numbers in the following order: the numerator n, the denominator d, and the accuracy epsilon.

The function is implemented in the Agena language and included in the `library.agn` file.

See also: **div** package.

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 is returned. The function is implemented in the Agena language and included in the `library.agn` file.

See also: **math.lcm**.

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.lcm (x, y)`

Returns the least common multiple of to numbers `x` and `y` as a number. The function is implemented in the Agena language and included in the `library.agn` file.

See also: math.gcd.

`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.ndigits (x)`

Returns the number of digits in the integral part of the number `x`.

`math.nextafter (x, y)`

Returns the next machine floating-point number of `x` in the direction toward `y`.

`math.nextprime (x)`

Returns the smallest prime greater than the given number `x`.

See also: 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, stats.scale.

`math.prevprime (x)`

Returns the largest prime less than the given number `x`.

See also: math.nextprime, math.isprime.

`math.Phi`

The golden number, Phi $:= \frac{1+\sqrt{5}}{2}$.

`math.random ([m [, n]])`

This function creates random numbers.

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`].

`math.randomseed (x, y)`

Sets `x` and `y` as the "seeds" for the pseudo-random generator: equal seeds produce equal sequences of numbers. `x` and `y` must both be positive integers. It returns two new settings.

`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. The function is implemented in the Agena language and included in the `library.agn` file.

See also: clock.todec .

`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.

7.9 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 **readlib** or **with** functions , e.g. `readlib 'mapm'`.

In OS/2, Haiku, and DOS, the package is built into the binary executable and does not need to be activated with **readlib**.

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.

By default, the precision is set to 17 digits, but you can change this any time with the **mapm.xdigits** function, e.g.:

```
> mapm.xdigits(100);
```

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.xlog10**	common logarithm
mapm.xarcsin	inverse sine	**mapm.xmul**	multiplication
mapm.xarcsinh	inverse hyperbolic sine	**mapm.xpow**	power
mapm.xarctan	inverse tangent	**mapm.xsign**	sign
mapm.xarctan2(x, y)	4 quadrant inverse tangent	**mapm.xsin**	sine
mapm.xarctanh	hyperbolic inverse tangent	**mapm.xsincos**	sine and cosine
mapm.xcbrt	cubic root	**mapm.xsinh**	hyperbolic sine
mapm.xcos	cosine	**mapm.xsqrt**	square root
mapm.xcosh	hyperbolic cosine	**mapm.xsub**	subtraction
mapm.xdiv	division	**mapm.xtan**	tangent
mapm.xexp	exponential function	**mapm.xtanh**	hyperbolic tangent

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
-	`'__unm'`	unary minus
<	`'__lt'`	less-than
=	`'__eq'`	equals
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
mapm.xdigits	sets the number of digits used in all subsequent calculations. With no argument, returns the current setting	**mapm.xtonumber**	converts an arbitrary precision number to an Agena number
mapm.xdigitsin	significant digits	**mapm.xtostring**	converts an arbitrary precision number to a string

7.10 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 **readlib** or **with** functions, e.g. `readlib 'calc'`.

A typical example might look like this:

```
> readlib 'calc'
```

Define a function f :=x → sin(x):

```
> f := << x -> sin(x) >>
```

Determine all its zeros over [-5, 5]:

```
> calc.zero(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, π]:

```
> calc.gtrap(f, 0, Pi):
1.9999999938721
```

`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.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.

See also: calc.interp, calc.clampedsplinecoeffs, 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.

See also: calc.clampedspline.

calc.dawson (x)

Computes Dawson's integral for a number `x`. The return is a number.

See also: expx2.

calc.dilog (x)

Computes the dilogarithm function for a number `x`. The return is a number.

calc.diff (f, x [, eps])

Computes the value of the first differentiation of a function `f` at a point `x`. If `eps` is not passed, the function uses an accuracy of the value stored to **Eps**. You may pass another numeric value for `eps` if necessary.

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

See also: **calc.xpdiff**.

calc.Ei (x)

Computes the exponential integral for a positive number `x`. The return is a number, and **undefined** if $x \leq 0$.

calc.fprod (f, a, b)

Computes the product of `f`(a), `...`, `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 $C(x) = \int_0^x \cos(\frac{\pi}{2}\, t^2)\, dt$ and returns it as a number.

calc.fresnels (x)

Computes the Fresnel integral $S(x) = \int_0^x \sin(\frac{\pi}{2}\, t^2)\, dt$ and returns it as a number.

calc.fseq (f, a, b [, step])

This function has been deprecated. Use **nseq** instead.

calc.fsum (f, a, b)

Computes the sum of `f`(a), `...`, `f`(b), with `f` a function, `a` and `b` numbers. If `a` > `b`, then the result is 0. The function uses Kahan round-off error prevention.

See also: **calc.fprod**.

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.

The function is implemented in Agena and included in the `lib/calc.agn` file.

See also: **calc.intde**, **calc.intdei**, **calc.intdeo**, **calc.integral**.

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.gtrap**, **calc.intdei**, **calc.intdeo**, **calc.integral**.

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.intdei**, **calc.integral**.

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.intdeo**, **calc.integral**.

calc.integral (f, a, b [, omega [, eps])

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.

See also: **calc.gtrap**, **calc.intde**, **calc.intdei**, **calc.intdeo**.

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, 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 itself in each call.

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.clampedspline**, **calc.nakspline**, **calc.neville**, **calc.newtoncoeffs**.

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`], ···, [`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` <= **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.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`], ···, [`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` <= **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.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.

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**.

See also: **calc.nakspline** .

calc.neville (obj, a)

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.

The function returns the value of the interpolating polynomial, a number, at the specified value `a` of the independent variable.

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.polygen (c_n, c_{n-1}, ···, c_2, c_1)

Creates a polynomial $p(x) = c_n*x^{n-1} + c_{n-1}*x^{n-2} + \cdots + c_2*x + c_1$ from the coefficients c_n, c_{n-1}, ···, c_2, c_1 and returns it as a new function p := << x-> p(x) >>, where x and the return p(x) represent numbers.

calc.Psi (x)

Computes the Psi (digamma) function (the logarithmic derivative of the gamma function) for a number `x`. The return is a number.

calc.Shi (x)

Computes the hyperbolic sine integral and returns it as a number. `x` must be a number.

See also: **calc.Ci**, **calc.Chi**, **calc.Si**, **calc.Ssi**.

calc.Si (x)

Computes the sine integral and returns it as a number. `x` must be a number.

See also: **calc.Ci**, **calc.Chi**, **calc.Shi**, **calc.Ssi**.

`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.xpdiff (f, x, [eps, [delta]])`

Like **calc.diff**, but uses Richardson's extrapolation method. `f` is the function to be iterated at point `x` (a number). `eps` and `delta` are accuracy values (numbers, as well). The return of the procedure are the derivative of `f` at `x` - a number - and the error.

xpdiff produces better results with powers and trigonometric functions than **calc.diff**.

`calc.zero (f, a, b, [step [, eps]])`

Returns all roots of a function `f` in one variable on the interval [`a`, `b`].

The function divides the interval [`a`, `b`] into smaller intervals [`a`, `a+step`], [`a+step`, `a+2*step`], `...`, [`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.

7.11 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 **readlib** or **with** functions, e.g. `readlib '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.

A sample session:

```
> with 'linalg'
linalg v1.0.2 as of February 13, 2011

add, augment, backsubs, checkmatrix, checksquare, checkvector, coldim,
column, crossprod, det, diagonal, dim, dotprod, hilbert, identity, inverse,
isantisymmetric, isdiagonal, isidentity, ismatrix, issquare, issymmetric,
isvector, ludecomp, matrix, mmap, mmul, mzip, rowdim, scalarmul, setvelem,
stack, sub, swapcol, swaprow, transpose, vector, vmap, vzip, zero
```

Define two vectors in two fashions: In the simple form, just pass all components explicitly:

```
> 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 <u>+</u> 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 <u>-</u> 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

multiplies a number `k` with each element in vector or matrix `s`. The return is a new vector or matrix. This operation is done by applying the `__mul` metamethod.

abs (v)

Determines the length of vector `v`. This operation is done by applying the `__abs` metamethod to `v`.

qsadd (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 `__qsadd` 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.backsubs (A, b)

Solves the set of linear equations `A*x = b`, where `A` is a matrix, and `b` the right-hand side vector. The return is the solution vector `x`.

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.

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.

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.

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.

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.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.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.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 sequence, 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.

The result is undefined if A is a singular matrix.

`linalg.matrix (`$\mathbf{obj_1, obj_2, \cdots, obj_n}$**`)`**

`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.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.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 nth root of the sum of the magnitudes (absolute values) of each element in v raised to the nth power.) If n is infinity, the return is the infinity norm, i.e. the maximum magnitude of all elements v.

`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.scalarmul (v, n)`

Performs a scalar multiplication by multiplying each element in vector v with 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.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.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.sub (v, w)`

Subtracts vector `w` from vector `v`. The result is a new vector.

See also: **linalg.add**.

`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` is 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.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.**v**zip**, **linalg.mmap**, **linalg.m**zip.

`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.zero (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, [])`.

7.12 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 **readlib** or **with** functions, e.g. `readlib 'stats'`.

You might want to use **utils.readcsv** to read distributions from a file.

`stats.ad (obj)`

Computes the absolute (or mean) deviation of all the values in a table or sequence `obj`, i.e. the mean of the equally likely absolute deviations from the mean m:

$$\frac{1}{n}\sum_{i=1}^{n}|obj_i - m|$$

The return is a number. Absolute deviation is more robust than standard deviation since it is less sensitive to outliers.

The function returns **fail** if `obj` contains less than two elements.

See also: **stats.sd**.

`stats.amean (obj)`

Divides each element in a table or sequence `obj` by the size of `obj` and sums up the quotients to finally return the arithmetic mean. It is equivalent to:

$$\sum_{i=i}^{n}\frac{obj_i}{n}$$

By dividing each element before summation, the function avoids arithmetic overflows and also uses a modified Kahan algorithm developed by Kazufumi Ozawa published in his paper `Analysis and Improvement of Kahan's Summation 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. With an empty table or sequence, the function returns **fail**.

The function also accepts structures including the value **undefined**. In this case, all **undefined**s are ignored, so that the function can be used with incomplete observations.

See also: **stats.gmean**, **stats.hmean**, **stats.mean**, **stats.qmean**, **stats.sma**.

stats.cdf (a, b [, μ [, σ]])

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:

$$\frac{1}{\sigma\sqrt{2\pi}} \int_a^b e^{\frac{-(x-\mu)^2}{2\sigma^2}}$$

See also: **stats.nde**, **stats.ndf**, **stats.pdf**.

stats.colnorm (obj)

Returns the largest absolute value of the numbers in the table or sequence `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** is 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 or sequence `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.

The function also accepts structures including the value **undefined**. In this case, all **undefined**s are simply included in the resulting structure and are ignored in the computation of the sums. Thus the function can be used with incomplete observations.

See also: **sadd**, **calc.fsum**, **stats.sum**.

`stats.deltalist (obj [, option])`

Returns a structure of the deltas of neighbouring elements in the table or sequence `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.

`stats.gmean (obj)`

Returns the geometric mean of all numeric values in table or sequence `obj` as a number. It is a measure of central tendency. Its formula is:

$$\left(\prod_{i=1}^{n} \mathrm{obj}_i\right)^{1/n}$$

With an empty table or sequence, the function returns **fail**.

The geometric mean should be applied on positive values that are interpreted to their products, e.g. rates of growth, instead of their sums.

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 observations.

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 observations.

stats.hmean (obj)

Returns the harmonic mean of all numeric values in table or sequence `obj` as a number. It is useful with rates and ratios, as it provides the truest average. It is defined as follows:

$$n \, / \sum_{i=1}^{n} \frac{1}{obj_i}$$

The function returns **fail** if `obj` is empty.

The harmonic mean should be applied on observations containing relations to a unit, e.g. speed.

The function is implemented in Agena and included in the `lib/stats.agn` file.

See also: **stats.amean**, **stats.gmean**, **stats.mean**, **stats.qmean**.

stats.ios (obj)

Sums up absolute differences between neighbouring entries in a table or sequence `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.

This indicator is quite useful to find out how stable or volatile a preferably unsorted observation is.

See also: **stats.ad**, **stats,sd**, **stats.var**.

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. **iqr** is useful for determining the variability in a distribution `obj` (a table or sequence).

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

stats.percentile (`obj`, `b`) - **stats.percentile** (`obj`, `a`)

The function is implemented in Agena and included in the `lib/stats.agn` file.

See also: **stats.percentile**, **stats.quartiles**.

`stats.issorted (obj [, f])`

Checks whether all values in a table or sequence `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.median (obj)`

Returns the median of all numeric values in table or sequence `obj` as a number. If `obj` is unsorted, it automatically sorts it before determining the median.

If `obj` is empty or entirely consists of **undefined**s, **fail** is returned. The function also ignores **undefined**s, if `obj` features numbers.

The median is the middle element of an observation if its size is odd, or the average of its middle elements it is size is even.

`stats.mean (obj)`

Returns the arithmetic mean of all numeric values in table or sequence `obj` as a number. It is equivalent to:

$$\frac{1}{n}\sum_{i=1}^{n} obj_i$$

If `obj` is table, it is assumed to be an array, non-positive integral keys (including strings, etc.) are ignored. With an empty table or sequence, the function returns **fail**.

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.amean**, **stats.gmean**, **stats.hmean**, **stats.qmean**.

`stats.minmax (obj [, 'sorted'])`

Returns a table with the minimum of all numeric values in table or sequence `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.

`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]])`

Computes the moment `p` of the given table or sequence `obj` about any origin x_m 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 x_m defaults to 0. If given, the moment `p` and the origin x_m must be numbers. If `obj` is empty, **fail** is returned.

See also: **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.ndf**, **stats.pdf**.

stats.ndf ([σ])

Computes $\frac{1}{\sqrt{2\pi}}$ if σ is not given, and $\frac{1}{\sigma\sqrt{2\pi}}$ otherwise, and issues an error if $\sigma \leq 0$.

See also: **stats.nde**, **stats.pdf**.

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.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 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)

Divides a numeric range defined by the pair `p` and its step size `n` into its subintervals, sorts all occurrences in the observation `s` (a sequence) into these subranges and finally counts all elements in these subranges.

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 is 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`.

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]
```

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 $\sigma = 1$. 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.percentile (obj, p [, option])

Returns the value below which a certain percent `p` of the elements in `obj` fall.

`obj` must be a table or sequence, `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. If `option` is the string `'nist'`, then the method proposed by NIST is used; if the string `'excel'` is given for `option`, then the algorithm used by Excel is used.

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 or sequence `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`.

The function is implemented in Agena and included in the `lib/stats.agn` file.

stats.qmean (obj)

Returns the quadratic mean of all numeric values in table or sequence `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 is equivalent to:

$$\sqrt{\frac{1}{n}\sum_{i=1}^{n} obj_i^2}$$

With an empty table or sequence, the function returns **fail**.

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 [, pos])`

Returns the first, second, and third quartile of a *sorted* table or sequence `obj`.

If only one argument is given, all three quartiles are returned. The type of return is determined by the type of `obj`.

If either the integer 1, 2, or 3 is passed for the optional second argument `pos`, the first second, or third quartile is returned as a number, respectively.

The number of values in `obj` should be at least 12, better are 20 or more values.

The function is implemented in Agena and included in the `lib/stats.agn` file.

See also: **whereis**, **stats.percentile**.

`stats.rownorm (obj)`

Returns the sum of the absolute values of the numbers in the table or sequence `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** is returned.

See also: **stats.scale**, **stats.colnorm**.

`stats.scale (obj)`

The procedure normalises the numbers in the table or sequence `obj` in such a way that an element of maximum absolute value equals 1. The normalised numbers are returned in a new table or sequence, depending on the type of `obj`.

See also: **math.norm**, **linalg.scale**.

stats.sd (obj)

Returns the standard deviation of all numeric values in table or sequence `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. It is described by the formula:

$$\sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(obj_i - m\right)^2}$$

If the return is a small number, it indicates that the points in an observation 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.

With an empty table or sequence, the function returns **fail**.

The function is implemented in Agena and included in the `lib/stats.agn` file.

See also: **stats.ad**, **stats.ssd**, **stats.var**.

stats.skewness (obj)

Returns the sample skewness, a measure of the asymmetry of the probability distribution of the numeric values in the table or sequence `obj`; returns 0 if a distribution is symmetric; a negative value if the left tail is longer; and a positive value the right tail is longer.

It computes the third moment about the mean and divides it by the third power of the standard deviation. If `obj` is empty, **fail** is returned.

The function is implemented in Agena and included in the `lib/stats.agn` file.

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 a modified Kahan algorithm developed by Kazufumi Ozawa to prevent round-off errors during summation.

stats.gsma is the iterator version of this function which traverses large observations 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 or sequence `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**.

stats.gsmm is the iterator version of this function which traverses large observations much faster.

See also: **stats.amean**, **stats.gsma**, **stats.gsmm**, **stats.sma**.

`stats.sorted (obj [, true])`

Sorts the table or sequence `obj` of numbers in ascending order and non-destructively up to 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. You may use an iterative variant of the Quicksort algorithm by passing the second argument **true**, which may be faster on some older systems, especially with elements in completely random or in (nearly) ascending order.

See also: **sort**, **sorted**, **skycrane.sorted**, **stats.issorted**.

`stats.ssd (obj)`

Returns the sample standard deviation of all numeric values in table or sequence `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. It is described by the formula:

$$\sqrt{\frac{1}{n-1}\sum_{i=1}^{n}\left(obj_i - m\right)^2}$$

The function returns **fail** if `obj` is empty or contains just one element.

The function is implemented in Agena and included in the `lib/stats.agn` file.

See also: **stats.ad**, **stats.sd**, **stats.var**.

`stats.sum (obj)`

`stats.sum (f, obj [,···])`

Sums up all the values of the given table or sequence `obj` and returns the sum (a number). Contrary to the **sadd** operator, it prevents round-off errors during summation. It is equivalent to:

$$\sum_{i=1}^{n} obj_i$$

In the second form, 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 `obj`.

stats.sum also accepts structures including the value **undefined**. In this case, all **undefined**s are simply ignored and skipped in the computation of the sums. Thus the function can be used with incomplete observations.

Examples:

```
> readlib 'stats';

> stats.sum(<< x -> x > 2 >>, seq(1, 2, 3, 4)):
7
```

```
> stats.sum(<< x, y -> x+y > 2 >>, seq(1, 2, 3, 4), 1):
9
```

See also: **sadd**, **calc.fsum**, **stats.cumsum**.

`stats.sumdata (obj [, p [, xm]])`

Sums up all the powers `p` of the given table or sequence `obj` of n elements about the origin x_m and returns a number. It is equivalent to:

$$\sum_{i=1}^{n} (obj_i - x_m)^p$$

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**.

See also: **stats.moment**.

`stats.tovals (obj)`

Converts all string values in the structure `obj` to Agena numbers. The type of return is determined by the type of `obj`.

`stats.var (obj)`

Returns the variance of all numeric values in table or sequence `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. The variance is defined as follows:

$$\frac{1}{n}\sum_{i=1}^{n} (obj_i - m)^2$$

With an empty table or sequence, the function returns **fail**.

The function is implemented in Agena and included in the `lib/stats.agn` file.

See also: **stats.ad**, **stats.sd**.

7.13 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.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.

Miscellaneous:

io.isfdesc, io.fileno, io.isopen, io.nlines, io.tmpfile.

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 then do od; # wait until a key has been pressed
```

The function works in the Solaris, Linux, Lion, 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.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.

See also: **io.open**, **io.popen**.

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.getkey ()

Waits until a key is pressed and returns its ASCII number.

The function is available in the Solaris, Linux, Mac OS X, and Windows editions only.

See also: **io.anykey**, **io.read**.

io.infile (filename, pattern)

Checks whether the file given by the name `filename` includes a `pattern` of type string, and returns **true** or **false**.

See also: **io.readfile**.

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.

The function is five times slower than **io.fdesc**.

See also: **io.fileno**, **io.isfdesc**.

`io.lines (filename)`

`io.lines (filehandle)`

`io.lines ()`

In the first form, the function opens the given file denoted by `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.

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.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 mode (the default);
- **'w'**: write mode only; if the file already exists, it is truncated to zero length;
- **'a'**: 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 (if `mode` is `'r'`, or `mode` is not given) via stdout, 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 the Agena language, 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: **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
```

This function is system dependent and is not available on all platforms.

See also: os.execute, io.pcall.

`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.
- '*a': reads the whole file, starting at the current position. On end of file, it returns the empty string[23].
- '*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 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. This keyboard input functionality is not available in AgenaEdit.

See also: io.lines, io.readfile, io.readlines, skycrane.readcsv, utils.readcsv, utils.readxml.

`io.readfile (filename [, true [, pattern [, flag]]])`

Reads the entire file with name `filename` 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, 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 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 not supported.

[23] See also io.readfile to read a file entirely.

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.read**, **io.readlines**, **io.writefile**.

`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. The function automatically deletes carriage returns (ASCII code 13) if included in the file.

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.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, function **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.skiplines**.

`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 `filehandle`. In the second form, the function flushes the default output.

`io.tmpfile ()`

Returns a handle for a temporary file. This file is opened in update mode and it is automatically removed when the programme ends.

`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. For more information, see io.lock.

io.write (···)

io.writeline (···)

Writes 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 or numbers. To write other values, use **tostring** or **strings.format**.

writeline adds a new line at the end of the data written, whereas **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 function with `<str>` being a string of any length. Remember that in the function call, a shortcut to `'delim':<str>` is `delim ~ <str>`.

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
```

See also: **io.writefile**, **print**, **skycrane.scribe**, **skycrane.tee**.

`io.writefile (fn, ···)`

Creates a new file `fn` 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.

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**.

7.14 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.

Reading data:

binio.readbytes, binio.readchar, binio.readlong, binio.readnumber, binio.readshortstring, binio.readstring.

Writing data:

binio.writebytes, binio.writechar, binio.writelong, binio.writenumber, binio.writeshortstring, binio.writestring.

File positions:

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 is 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
```

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.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.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.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.

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 [, anything])

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.

If the file already exists, it leaves it open 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.) Use binio.toend to append to the file.

The file is always opened in both read and write modes.

If an optional second argument is given (any valid Agena value), 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.

binio.readbytes (filehandle [, bytes])

In the first form, the function reads environ.buffersize bytes from the file denoted by `filehandle` and returns them as a sequence of integers. You may change the value of environ.buffersize to any other positive integer in order to read less or more bytes. The default for environ.buffersize is 512 bytes.

In the second form, 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 is returned. In case of an error, it quits with this error.

The function is much faster when working on a larger number of bytes.

See also: binio.writebytes, strings.toBytes.

binio.readchar (filehandle)

binio.readchar (filehandle, position)

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 `position` 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 is returned as a number. If the end of the file has been reached, **null** is returned. In case of an error, the function quits.

binio.readlong (filehandle)

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 an error or nothing to read, the function quits with an error. Note that the number to be read should have been written to the file using the **binio.writelong** function.

See also: **binio.writelong**.

binio.readnumber (filehandle)

The function reads an Agena number from the file denoted by `filehandle` from the current file position and returns it. If there is an error or nothing to be read, the function quits with an error. Note that the number to be read should have been written to the file using the **binio.writenumber** function.

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 an error or nothing to read, the function quits with an error.

Note that the string to be read should have been written to the file using the **binio.writeshortstring** function, as **binio.writeshortstring** also stores the length of the string 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 an error or nothing to read, the function quits with an error.

Note that the string to be read should have been written to the file using the **binio.writestring** function, as **binio.writestring** also stores the length of the string to the file.

See also: **binio.writestring**.

binio.rewind (filehandle)

Sets the file position to the beginning of the file denoted by `filehandle`. 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. 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 <= `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**.

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 <= `number` < 256, otherwise `number` % 256 will be stored to the file.

Internally, the bytes are stored as a C `unsigned char`.

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.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** .

7.15 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 **readlib** or **with** function, e.g. `readlib 'xbase'`.

This package provides basic functions to read and write dBASE III-compliant files.

A typical session may look like this:

```
> with 'xbase'

> 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: Numbers, Strings, and Logical.
2. Only files with extension .dbf are supported. Searching and sorting functions are not available, and any .ndx or .idx index 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, 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.

See also: **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'])

Returns all values in the given field `row` (a number) of the file denoted by `filehandle` and by default returns them in a sequence. If the optional third argument `'set'` is given, the return will be a set of all the values in the field.

See also: **xbase.readdbf**, **xbase.readvalue**, **xbase.record**.

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.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.readvalue**, **xbase.purge**.

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.

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.new (filename, $desc_1$ [, codepage] [, $desc_2$, ···, $desc_k$])
creates a new xBase file with the file name `filename`.

$desc_k$ are `k` fields (columns) the xBase file will have. codepage indicates the code page to be used (see below)[24].

$desc_k$ 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 `'boolean'`, `'number'`, or `'string'`, i.e. the data type of the values to be entered later.

   Examples:

   `new('dbase.dbf', 'logical':'boolean');` or
   `new('dbase.dbf', logical=boolean);` for short.

   A Boolean (which in xBase has the synonym `Logical`) will always consist of one character `'T'`, `'F'` for **true** and **false**.

   A number will have a standard length of 19 places with a default scale of 15 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].

   A string will have the 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.

   Examples:

   `new('dbase.dbf', 'value':'number':0);` or
   `new('dbase.dbf', value=number:5);` for short.

2. 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.

   You may leave off the quotes for data_type values.

`codepage` should be a pair of the form `'codepage'`:n, with n an integer in [0, 255].

[24] Note that code pages are a Foxpro extension.

Valid codepages are:

n	Meaning	Code page
0x01	DOS USA	437
0x02	DOS Multilingual	850
0x03	Windows ANSI	1.252
0x04	Standard Macintosh	
0x64	EE DOS	852
0x65	Nordic DOS	865
0x66	Russian DOS	866
0x67	Icelandic DOS	
0x68	Kamenicky (Czech) DOS	
0x69	Mazovia (Polish) DOS	
0x6a	Greek DOS	437G
0x6b	Turkish DOS	
0x96	Russian Macintosh	
0x97	Eastern European Macintosh	
0x98	Greek Macintosh	
0xc8	Windows EE	1.250
0xc9	Russian Windows	
0xca	Turkish Windows	
0xcb	Greek Windows	

If no code page has been passed, it is set to 0x00.

Example for Russian Macintosh:

```
new('dbase.dbf', text=string:255, codepage=0x96);
```

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'`, `'append'`, or `'r+'`, the file is opened for reading while new data sets may be added at 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.new, xbase.lock.

xbase.purge (filehandle, record, field)

Marks the specific `field` in the given `record` of the file denoted by its handle `filehandle` as deleted. The return is **true** if deletion succeeded, and **false** otherwise.

See also: **xbase.isVoid**.

xbase.readdbf (filename)

Opens an xBase file denoted by its `filename` in read mode, returns all its records and fields, and closes it.

If the xbase file contains more than one field, the data is returned as a sequence of sequences, whereas if the file contains only one field, all values are returned in one sequence.

See also: **xbase.readvalue**, **xbase.field**, **xbase.field**.

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, and String. If a number could not be read from the file, the function returns 0.

See also: **xbase.field**, **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.

See also: **xbase.readdbf**, **xbase.readvalue**. **xbase.field**.

xbase.sync (filehandle)

Writes any unwritten content to the xBase file pointed to by `filehandle`. The function either returns **true** if flushing succeeded, or **fail** otherwise.

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. For more information, see **xbase.lock**.

`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.

The return is **true** if writing succeeded, and **false** otherwise.

`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 return is **true** if writing succeeded, and **false** otherwise. Note that the return **false** only indicates that an error may have occurred.

`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`.

The return is **true** if writing succeeded, and **false** otherwise. Note that the return **false** only indicates that an error might have occurred.

7.16 xml - XML Parser

As a *plus* package, the **xml** package is not part of the standard distribution and must be activated with the **readlib** or **with** functions, e.g. `readlib 'xml'`. It is available for Solaris, 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.

7.16.1 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.

7.16.2 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.

7.16.3 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 the Agena language and included in the `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.readxml**, **utils.readcsv**, **xml.decode**, **xml.decodexml**.

7.16.4 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.

7.16.5 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)`

Set 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".

7.16.6 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 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'.

7.17 gzip - Library to Read and Write UNIX gzip Compressed Files

As a *plus* package, in Solaris, Linux, Mac OS X, DOS, and Windows, this library is not part of the standard distribution and must be activated with the **readlib** function, , e.g. `readlib 'gzip'`.

The package is not available in Haiku and OS/2.

A typical session may look like this:

```
> readlib '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.flush (filehandle)`

This function takes a file handle and flushes all output to the working file.

`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, value`$_1$`, ···)`

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**.

7.18 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 **readlib** function, e.g. `readlib 'net'`.

The package is not available in DOS, Haiku, and OS/2.

7.18.1 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.

```
> with 'net'
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[25] 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 is 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
```

[25] 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:

```
> with 'net'
```

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.

<table>
<tr><th>Server</th><th>Client</th></tr>
<tr><td><pre>
> with 'net'

> 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></td><td><pre>
> with 'net'

> d := open():
352

> connect(d, 'localhost', 1300):
true

> send(d, strings.repeat('#', 1m)):
1000000

> close(d):
true
</pre></td></tr>
</table>

A simple bi-directional connection:

<table>
<tr><th>Server</th><th>Client</th></tr>
<tr><td><pre>
> with 'net'

> 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></td><td><pre>
> with 'net'

> 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></td></tr>
</table>

<table>
<tr><th>Server</th><th>Client</th></tr>
<tr><td><pre>> while y <> 0 do
>   x, y := receive(e);
>   print(x, y);
>   send(e, 'Got ' & y & '
bytes');
> od;
etc.   488
        0

> close(e, d):
true</pre></td><td></td></tr>
</table>

Usage of black and whitelists: First initialise the **net** package.

```
> with 'net';
```

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():
[]
```

7.18.2 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[26]. 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**.

`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

[26] 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.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.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.

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 can be determined or set by reading or setting the value of **environ.buffersize** which value depends on the operating system.

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.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 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**.

7.19 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.exists**, **os.fattrib**, **os.fcopy**, **os.fstat**, **os.list**, **os.listcore**, **os.mkdir**, **os.move**, **os.readlink**, **os.remove**, **os.rmdir**, **os.symlink**, **os.tmpname**.

Hardware access:

os.battery, **os.beep**, **os.endian**, **os.freemem**, **os.memstate**, **os.mousebuttons**, **os.screensize**.

Operating System Access:

os.computername, **os.cpuinfo**, **os.cpuload**, **os.drives**, **os.drivestat**, **os.environ**, **os.execute**, **os.exit**, **os.getenv**, **os.isANSI**, **os.isUNIX**, **os.login**, **os.pid**, **os.setenv**, **os.settime**, **os.setlocale**, **os.system**, **os.wait**.

Date and Time:

os.date, **os.datetosecs**, **os.difftime**, **os.now**, **os.secstodate**, **os.time**.

`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
`'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)

Key	Meaning
'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, 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` (a positive integer) for `dur` seconds (a positive float) in Windows and OS/2. In UNIX and DOS, 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.chdir ([str])`

Changes into the directory given by string `str` on the file system. Returns **true** on success and issues an error on failure otherwise. If no argument is given or **null** is passed for `str`, the name of the current working directory is returned as a string.

`os.computername ()`

Returns the name of the computer in Windows, DOS, Mac OS X, Haiku, and UNIX. The return is a string. On other architectures, the function returns **fail**.

`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, DOS, Linux, and Mac OS X only[27]. The return is a table with the following fields:

[27] In Solaris, you may issue `io.pcall('kstat')` and parse its return.

Field	Meaning	Windows	Mac	Linux
`'bigendian'`	endianness: **true** means Big Endian, **false** Little Endian, and **fail** undetermined.	x	x	x
`'brand'`	processor name, a string[28]	x	x	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	
`'revision'`	processor revision, a posint	x		
`'stepping'`	processor stepping, a posint		x	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
`'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.

If executed on systems other than Windows, Linux, DOS, Sparcs, and Mac OS X, the function returns **fail**.

The Linux version has been written in the Agena language, see the `library.agn` file; the other OS versions have been implemented in C.

See also: **os.cpuload**, **os.endian**.

`os.cpuload ()`

Returns the 1, 5 and 15 minute load averages of the computer as a sequence of three numbers in the range [0 , 1]. It is available in Linux and Mac OS X only.

See also: **os.cpuinfo**.

`os.curdir ()`

Has been deprecated. Please use **os.chdir(null)** to determine the current working directory.

[28] The return may include leading or trailing blanks.

os.date ([format [, time]])

Returns a string or a table containing date and time, formatted according to the given string `format`.

If the time argument is present, this is the time to be formatted (see the **os.time** function for a description of this value). Otherwise, **date** formats the current time.

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, Sunday is 1), `yday` (day of the year), and `isdst` (daylight saving flag, a boolean).

If `format` is not `*t`, 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'.

See also: **os.now**, **os.time**.

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 or sequence `obj` being integers, and transforms it to the number of seconds elapsed since the start of an `epoch`.

In the second form, receives the given integers, and conducts the same operation.

The time zone acknowledged may depend on your operating system.

See also: **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.drives ()

In Windows and OS/2, the function returns all the logical drives available at the local computer. The return is a sequence of drive letters. In other systems, the return is **fail**.

os.drivestat (driveletter)

In Windows, the function returns information of the given logical drive (a single letter string) in a table where its keys have the following meaning:

Key	Meaning
`'label'`	the drive label
`'filesystem'`	the file system (e.g. NTFS, FAT32, 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

In other systems, the return is **fail**.

Example:

```
> os.drivestat('c'):  # get information on drive C:\
[filesystem ~ NTFS, label ~ drive_c, drivetype ~ Fixed, freesize ~
75547742208, totalsize ~ 85898014720]
```

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.execute ([command])

This function is equivalent to the C function `system`. It passes `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.

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**.

os.exit ([code])

Calls the C function `exit`, with an optional `code`, to terminate the host programme. The default value for `code` is the success code.

os.fattrib (fn, mode)

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 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 time stamps 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', [2012, 05, 23, 12, 30, 0]); # sets time stamp
```

See also: **os.fstat**, **os.now**.

os.fcopy (infile, outfile)

Copies the file and its permissions denoted by the filename `infile` to the file called `outfile`. If `outfile` already exists, it is overwritten without warning. The function internally uses **environ.buffersize** for the number of bytes to be copied at the same time, which you may change to any other positive integer.

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.

Please note that `outfile` cannot specify a target directory. Use **skycrane.fcopy** instead which copies files into other files and also to directories.

See also: **skycrane.fcopy**.

os.freemem ([unit])

Returns the amount of free physical RAM available on Windows and Mac OS X, Haiku, 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 GBytes. On other architectures, the function returns **fail**.

See also: **environ.used**, **os.memstate**.

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
`'date'`	last modification date in the form yyyy, mm, dd, hh, mm, ss
`'perms'`	file attributes coded in an integer (C type file attributes as 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.

Key	Meaning
	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)
`'bits'`	The permission bits.
`'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.
`'blocksize'`	(UNIX only) Optimal block size for reading or writing this file, in bytes.
`'device'`	Device containing the file, in Windows 0 = A, 1 = B, etc.
`'inode'`	(UNIX only) Unique file serial number.

See also: **os.fattrib**.

`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.isANSI ()`

Returns **true** on Agena editions compiled with the LUA_ANSI (strict ANSI C) option, and **false** otherwise.

`os.isUNIX ()`

Returns **true** if Agena is being run in a UNIX environment (i.e. Solaris, Linux, and OpenSolaris), and **false** otherwise.

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.

`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 multiple subdirectories.

If no option is given, files, links, and directories are returned. If the optional argument `'files'` is given, only files are returned. If the optional argument `'dirs'` is given, directories are returned exclusively. If the optional argument `'links'` 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 the Agena language and included in the `library.agn` file.

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'`, `'dirs'`, or `'links'`, 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 `?` and `*`. If given, the function only returns those filenames 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**.

os.memstate ([unit])

(Windows, UNIX, Mac OS X, Haiku, 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.

The resulting table will contain the following values, an 'x' indicates which values are returned on your system.

Key	Description	Windows	UNIX/ Haiku	OS/2	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			
`'resident'`	occupied resident pages			x	
`'active'`	active memory				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.

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, Haiku, Mac OS X, and Windows based systems only.

`os.mousebuttons ()`

In Windows, returns the number of buttons of the attached mouse. If a mouse is not connected to your system, 0 is returned. On all other platforms, the function returns **fail**.

`os.move (oldname, newname)`

Renames or moves a file or directory named `oldname` to `newname`. The function returns **true** on success, and issues an error on failure otherwise.

See also: **skycrane.move** .

`os.now ([secs])`

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, information on this date and time are determined (see **os.datetosecs** to convert a date to seconds).

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 \`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**, **os.datetosecs**, **os.secstodate**, **os.time**.

`os.pid ()`

Returns Agena's process ID as a number.

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 not available in DOS.

See also: **os.symlink**.

os.remove (filename)

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.

os.rmdir (dirname)

Deletes a directory denoted by the string `dirname` on the file system. Returns **true** on success, and issues an error on failure otherwise.

os.screensize ()

In 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** is 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.setlocale (locale [, category])`

Sets the current locale of the programme. `locale` is a string specifying a locale; `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: **skycrane.getlocales** .

`os.settime (secs)`

Takes the number of seconds `secs` elapsed since the start of an epoch, in your local time zone, and sets the system clock accordingly. Agena must be run in root mode in order to change the system time. In case of an error, **fail** is returned. The function is only available in the Windows, Solaris, and Linux versions of Agena.

See also: **os.datetosecs** .

`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*), the minor version (*dwMinorVersion*), and the product type (*wProductType*) in this order.

In UNIX, Mac OS X, Haiku, OS/2, and DOS, it returns a table of strings with the name of the operating system (e.g. 'SunOS'), the release, the version, and the machine, in this order. Note that Mac OS X is recognised as 'Darwin'.

If the function could not determine the platform properly, it returns **fail**.

See also: **environ.os** .

`os.time ([table])`

Returns the current time when called without arguments, or a time representing the date and time specified by the given table. This table must have fields `year`, `month`, and `day`, and may have fields `hour`, `min`, `sec`, and `isdst` (for a description of these fields, see the **os.date** function).

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 `time` can be used only as an argument to **date** and **difftime**.

If a second number is returned, it denotes the millisecond portion of the current time in the range [0, 999].

See also: **time**, **os.date**, **os.difftime**, **os.now**.

`os.tmpname ()`

Returns a string with a file name that can be used for a temporary file. The file must be explicitly opened before its use and explicitly removed when no longer needed.

`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, Haiku, and Windows based systems only.

On other architectures, the function returns **fail**.

7.20 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 the Agena language and included in the `library.agn` file.

`environ.attrib (obj)`

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,
- 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 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,
- 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,
- 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 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,
- an indicator `'bytes'` stating the estimated number of bytes reserved,
- if present, a user-defined type is indexed by the `'utype'` key, otherwise **fail**.

With the pair `obj`, returns a new table with

- an indicator `'bytes'` stating the estimated number of bytes reserved,
- if present, a user-defined type is indexed by the `'utype'` key, otherwise **fail**.

`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.

`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.globals (f)`

Determines[29] whether function `f` includes global variables (names which have not been defined local).

`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 the Agena language and included in the `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 is returned.

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
`'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.
`'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.
`'gui'`	**true** or **false**	If set **true**, tells the interpreter that it has been invoked by AgenaEdit. Default is **false**.
`'libnamereset'`	**true** or **false**	If set **true**, the **restart** statement resets **libname** and **mainlibname** to their original values. Default is **false**.

[29] Note that the function not always returns all global names.

Setting	Value	Description
`'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**.
`'promptnewline'`	**true** or **false**	If set to **true**, prints an empty line between the input and outputline regions. Default is **false**.
`'signedbits'`	**true** or **false**	If set to **true**, the bitwise operators **&&**, **~~**, **||**, **^^**, and **shift** internally use signed integers (the default), otherwise they use unsigned integers.
`'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'):
true

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

`environ.pointer (obj)`

Converts `obj` to a generic C pointer (void*) and returns the result as a string. `obj` may be a userdata, table, set, sequence, pair, thread, function, or complex value; otherwise, **pointer** returns **fail**. Different objects will give different pointers.

`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.used ( [opt] )`

By default, returns the total memory in use by Agena in Kbytes. If `opt` is the string `'bytes'`, `'kbytes'`, `'mbytes'`, or `'gbytes'`, the number is 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 !
```

7.21 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.

`package.readlibbed`

A table with all the names of the packages that have been initialised with the **readlib** and **with** functions.

`package.register (packagename, name1 [, name2, ···])`

This function has been deprecated.

7.22 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.rdelete (f)`

Deletes the remember table or read-only remember table of procedure `f` entirely. The function returns **null**.

`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` is returned. See **rtable.rget** 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:

```
with('rtable', '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 the Agena language and included in the `library.agn` file.

See Chapter 6.18 for examples. See also: **rtable.defaults**.

`rtbale.rget (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 **rget**.

`rtbale.rinit (f)`

Creates a remember table (an empty table) for procedure `f`. The procedure must have been written in the Agena language; 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. **rinit** returns **null**.

`rtbale.rmode (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.

`rtbale.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**.

`rtbale.rset (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:

`rset(f, [1], [2])`; `rset(f, [1, 2], [2])`; `rset(f, [1], [1, 2])`.

7.23 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.

7.24 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.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.

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.

`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.setlocal**.

`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.

`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.setupvalue**.

`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.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 (n)`

Returns a table with the following system information: The size of various C types (char, int, long, long long, float, double, int32_t), the endianness of your platform, the hardware and the operating system for which the Agena executable has been compiled.

`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.

7.25 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 is returned. If `x` is invalid, the function issues **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.decodeb64 (str)`

Decodes the Base64 encoded string `str` and returns it as a string.

See also: **utils.encodeb64**.

`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 the Agena language 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.encodeb64 (str)`

Encodes a string `str` into Base64 format and returns it as a string.

See also: **utils.decodeb64**.

`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.findfile('*.c', '#define'):

> utils.findfile('*.c', << x -> '#define' in x = 1 >>, 'whole'):

> utils.findfile([['a.txt', 'b.txt'], 'c:/text'], 'hello'):
```

`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[30]. If a line contains only one field, then it is returned without including it in a sequence[31]. If a line contains nothing, i.e. '\n', it is by default ignored[32].

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**.	`delim = false`
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.	`delim = '|'`
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	`fields = [3, 1, 5]`

30 See the `flat` option to override this behaviour.

31 See the `newseq` option to override this behaviour.

32 See the `skipemptylines` option to override this behaviour.

Left pair element	Right pair element	Example

	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 = ['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`
ignorespaces	**true** or **false**: all spaces in a line are deleted before returning the fields. Default is **true**.	`ignorespaces = false`

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]`

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 is 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 is returned.	`output = 'record'`

Left pair element	Right pair element	Example
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.	`remove = 'quotes'`
skipemptylines	**true** or **false**: If **true**, do not return empty lines. Default is **true**.	`skipemptylines = true`
skipspaces	**true** or **false**: If **true**, do not return lines consisting of spaces only. Default is **false**.	`skipspaces = true`
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 the Agena language and included in the `libary.agn` file.

See also: **columns**, **io.lines**, **io.readlines**, **utils.readxml**, **utils.writecsv**, **skycrane.readcsv** .

`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.

utils.writecsv (obj, filename [, delim [, keyoption [, dot]]])

Creates a comma-separated value (CSV) file. The function writes all values or keys and value(s) of a table, set, or sequence `obj` to a text file given by `filename`. Each value or key ~ value pair is written on a separate line.

By default only values are written, the keys are ignored.

If the optional argument `delim` (a string) is given and if the value itself is a structure, then all entries in this substructure are written in a separate line, separated by the given delimiter; default is a semicolon. `delim` might also be of the form `delim=`<any string> or `delim`:<any string>.

If the optional argument `keyoption` is given, of any value other than **false**, **fail**, or **null**, then the key and the value(s) are also printed and are separated by the given delimiter (third argument) which must be passed, as well.

If the argument `dot` is given, i.e. a single character of type string, then a decimal dot in a number to be written is replaced by `dot`. This, for example, allows to replace a decimal dot in a float with a decimal comma. When wanting to substitute decimal dots, you must also pass either true or false for the fourth, `keyoption` argument.

The function returns nothing, is written in the Agena language and included in the `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='|', true,  ',');
```

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.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 the Agena language and included in the `library.agn` file.

See also: **utils.decodexml** , **utils.encodexml** , **utils.readxml**.

7.26 skycrane - Auxiliary Functions

As a *plus* package, the **skycrane** package is not part of the standard distribution and must be activated with the **readlib** or **with** functions, e.g. `readlib '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 is issued.

The function automatically loads the **bags** package if it has not yet been initialised.

The function is written in the Agena language and included in the `skycrane.agn` file.

See also: **bags.bag**.

`skycrane.count ([start [, step]])`

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 `start` or `step` are not numbers, the factory issues an error.

If `start` or `step` is a non-integer, the function automatically applies the Kahan summation algorithm to avoid round-off errors.

See also: **skycrane.iterate**.

`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, 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 the Agena language and included in the `skycrane.agn` file.

See also: os.fcopy, skycrane.move .

skycrane.getlocales ()

Returns all locales available at 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 .

Since the function has been implemented generically, it is very slow, for os.setlocale is called around 476.000 times. In UNIX, it would be better to issue the command 'locale -a' in a shell to determine the locales supported by your system.

The function is written in the Agena language and included in the `skycrane.agn` file.

See also: os.setlocale .

skycrane.iterate (o)

Returns an iterator function traversing a table, set, or sequence `o`.

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, 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 the Agena language and included in the `skycrane.agn` file. For the order how keys or values will be sorted, see **sorted**.

A note: This function is utterly slow compared with the **for/in** statement. There may be few situations demanding loops iterating in the strict ascending order of its (numeric or string) indices, or set and sequence values.

See also: **next**, **sorted**, **skycrane.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 the Agena language and included in the `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='quotes', remove='doublequotes'.

The function is written in the Agena language and included in the `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` is returned unmodified.

See also: **skycrane.enclose**.

skycrane.scribe (fh, obj [, ···])

skycrane.scribe (obj [, ···])

skycrane.scribe (···)

Like **io.write** and **io.writeline**, but if a table or sequence `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:

```
> readlib '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');

> paternoster[33] := seq(
>    '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 the Agena language and included in the `skycrane.agn` file.

See also: **io.write**, **io.writeline**, **skycrane.tee**.

[33] Taken from the Gothic Language Wulfila Bible edited by Wilhelm Streitberg.

skycrane.sorted (obj [, f])

Sorts a table or sequence `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 the Agena language and included in the `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 the Agena language and included in the `skycrane.agn` file.

skycrane.tee (fh, x [,···] [, 'delim':str])

Writes one or more numbers or strings `x` to both the console (stdout) and a file denoted by its handle `fh`. By default, the values are separated with a tabulator (`\t`). By passing the option `'delim'`:`str`, as the last argument, the delimiter is given by the string `str`.

The function returns nothing. See also: **print**, **skycrane.scribe**.

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.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` is returned unmodified.

7.27 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 **readlib** or **with** functions, e.g. `readlib '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 nonnegative, 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:

```
> with 'clock'
clock package v1.3.0 as of May 15, 2013

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 is 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.

7.28 astro - Astronomy Functions

As a *plus* package, the **astro** package is not part of the standard distribution and must be activated with the **readlib** or **with** functions, e.g. `readlib 'astro'`.

`astro.cdate (x)`

Converts a Julian date, represented by the float `x`, into its calendar date representation, returning three integer values: the year, the month, and the day, i.e. three integers.

See also: **astro.jdate**.

`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.isleapyear (x)`

Returns **true** if the given year `x` (a number) is a leap year, and **false** otherwise.

`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.

The defaults for `hour`, `minute`, and `second` are 0.

See also: **astro.cdate**.

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 the Agena language 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 the Agena language 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**.

7.29 ads - Agena Database System

As a *plus* package, this simple database is not part of the standard distribution and must be activated with the **readlib** or **with** functions, e.g. `readlib '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 very fast and stored to a set (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.

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:

```
> with 'ads';
```

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 processing. The variable `fh` is the file handle which references to the database file (`c:\test.agb`) and is used in 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 enter any entries. Use the **open** function if you want to have access again.

```
> closebase(fh);
```

On all 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

Key	Description	Type
'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

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** is returned. If all records could be deleted successfully, **true** is 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 is** issued, and no comment is written. **fail** is 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 the 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 20000. If you pass 0, fail is 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** is 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 the Agena language and can be used after running readlib 'ads'.

ads.desc (filehandle)
ads.desc (filehandle, description)

In the first form, returns the description of a base stored in the file header.

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** is returned and there are no changes to the base file. If the file is not open, **fail** is 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 - so the greater n, the faster shifting will be, 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.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)

Converts a sequence to a set and returns this set. The function automatically initialises the set with the number of entries in the sequence. If the file is not open, **fail** is returned.

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** is returned. If the base is empty, **null** is 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** is returned. If the base is empty, **null** is 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** is returned.

ads.indices (filehandle)

Returns the file positions of all valid detests 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 is returned. The function issues an error if the file contains a sequence.

See also: **ads.retrieve**, **ads.invalids**, **ads.peek**, **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)

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. If there are no more keys left, the function returns **null**. If the database is empty, **null** is 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);
```

`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.peek (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** is 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** is 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 is 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** is 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 reshifting). 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**.

7.30 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 **readlib** or **with** functions, e.g. `readlib '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.

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.

7.30.1 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.

```
> readlib('gdi');

> d := gdi.open(640, 480);
```

7.30.2 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);
```

7.30.3 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 explicitely 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**.

7.30.4 Closing a File or Window

To finally close the window, use **gdi.close**.

```
> gdi.close(d);
```

7.30.5 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.

7.30.6 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.

```
> with 'gdi'

> 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.

7.30.7 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).

```
> with 'gdi';
```

```
> 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`

7.30.8 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'`.

7.30.9 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 7.30.3), 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 7.30.3).

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 7.30.3. 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 7.30.3), 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 7.30.3).

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 7.30.3), 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 7.30.3).

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 <= x <= 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 7.30.3), may be given optionally.

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** 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'`

Option (key)	**Meaning (value)**	Example
'axescolour'	defines the colour of the axes (a colour string, see Chapter 7.30.6)	'axescolour':'red'
'bgcolour'	sets the background colour (a colour string, see Chapter 7.30.6)	'bgcolour':'yellow'
'colour'	sets the default colour (a string, see Chapter 7.30.6) 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** function 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
'thickness'	sets the thickness (a positive number) of the line to be plotted (**gdi.plotfn** function 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 7.19.6) of the title (**gdi.plotfn** function 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

See also: **gdi.setoptions** .

`gdi.point (d, x, y [, colour])`

Plots a point with co-ordinates [`x`, `y`] on device `d`. A `colour`, an integer (see Chapter 7.30.3), may be given optionally.

`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);
```

`gdi.plotfn (f, a, b [ [ c, d], options])`

`gdi.plotfn (ft, a, b [ [ c, d], options])`

Plots graphs of one or more functions.

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.

```
> with('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`.

```
> plotfn(<< x -> sin(x) >>, -10, 10, -2, 2);
```

Specify a colour other than black:

```
> plotfn(<< x -> sin(x) >>, -10, 10, colour='red');
```

Give a specific thickness for the line:

```
> plotfn(<< x -> sin(x) >>, -10, 10, thickness=3);
```

Combine the options - their order does not matter:

```
> plotfn(<< x -> sin(x) >>, -10, 10, thickness=3, colour='red');
```

Plot two and more functions:

```
> plotfn([<< x -> sin(x) >>, << x -> cos(x) >>], -10, 10);
```

Give options, too:

```
> 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:

```
> plotfn([<< x -> sin(x) >>, << x -> cos(x) >>], -10, 10,
>    colour=['red', 'cyan']);
```

Choose another colour for the axes and another axes style:

```
> plotfn([<< x -> sin(x) >>, << x -> cos(x) >>], -10, 10,
>    colour~['red', 'cyan'], axescolour='grey', axes='boxed');
```

Do not draw axes:

```
> 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');

> plotfn([<< x -> sin(x) >>, << x -> cos(x) >>], -10, 10)
```

See also: **calc.clampedspline**, **calc.nakspline**.

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 7.30.3), 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 7.30.3).

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])

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.

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])

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.

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])

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.

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 unconstrained = false option scales the x and y dimensions equally, the unconstrained = true 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, unconstrained = true);
```

gdi.setline (s, x1, y1, x2, y2 [, colour])

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.

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])

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.

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])

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.

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])

Draws a triangle with the corners [`x1`, `y1`], [`x2`, `y2`], and [`x3`, `y3`] on device `d`. A `colour` (an integer, see Chapter 7.30.3), may be given optionally for the lines.

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 7.30.3).

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.

7.31 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 **readlib** or **with** functions, e.g. `readlib 'fractals'`.

Since it needs **gdi** graphics functions, it is of no use in OS/2 and DOS.

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 7.31.4 for some examples.

7.31.1 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 * fractals.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 the Agena language.

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 the Agena language.

fractals.alcosxx (x, y, iter, radius)

This function calculates the Julia set of the formula lambda * fractals.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 the Agena language.

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 the Agena language.

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 the Agena language.

See also: fractals.newton .

fractals.lbea (x, y, iter, radius)

This function calculates the Julia set of the formula lambda * fractals.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.

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.

7.31.2 Auxiliary Mathematical Functions

fractals.bea (z)

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 meaningless, but it creates beautiful fractals.

fractals.cosxx (z)

Takes the complex number z = x!y and returns the complex number cos(x)*cosh(y)+I*sin(x)*sinh(y). It represents FRACTINT's buggy cos function till v16. This function may be mathematically meaningless, but it creates beautiful fractals.

fractals.flip (z)

Takes the complex number z and returns the complex number imag(z)!real(z).

7.31.3 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
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:

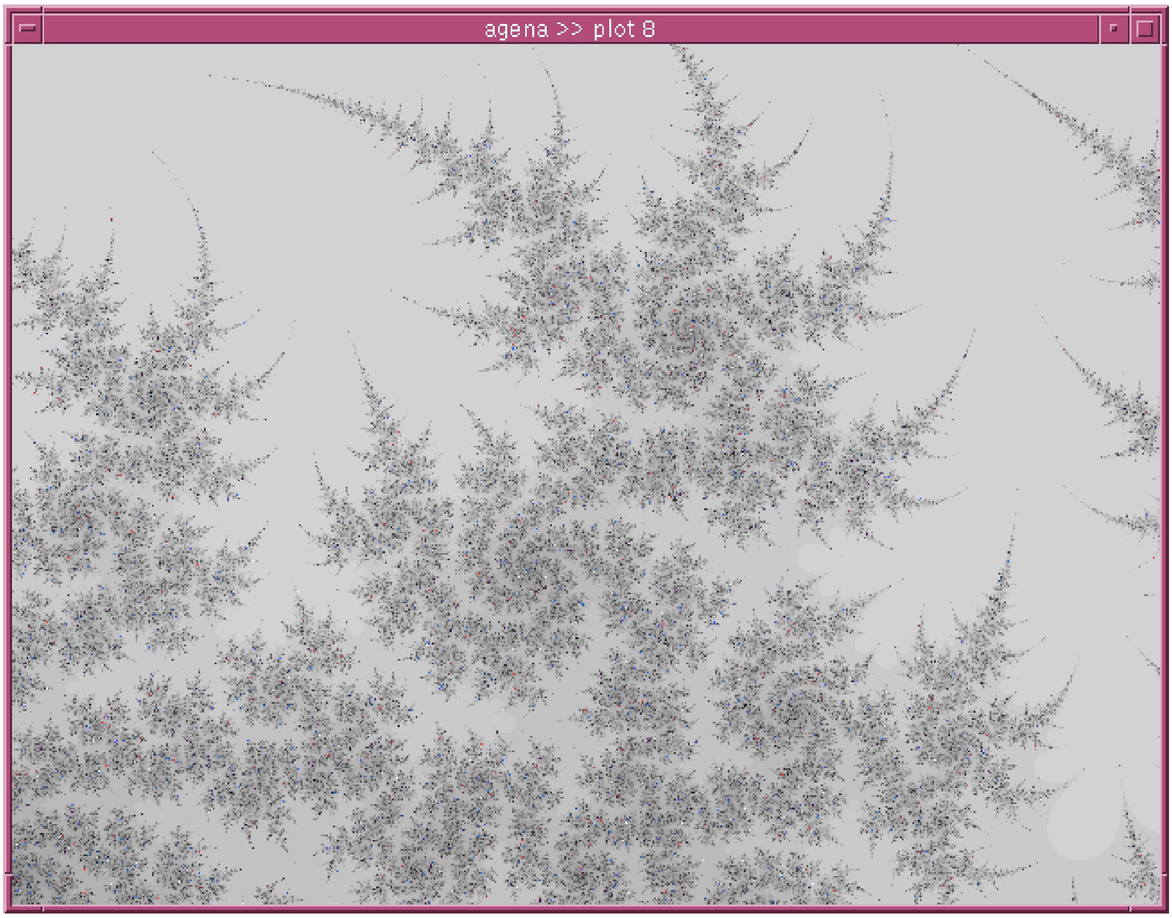
On all operating systems the default is 1. This behaviour can globally be changed in a session by assigning a non-negative integer to the environment variable **environ.fractscreenupdates**.

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.

7.31.4 Examples

```
> with 'fractals';

> 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 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
```

7.32 div - Library to Process Fractions

As a *plus* package, this library is not part of the standard distribution and must be activated with the **readlib** or **with** functions, e.g. `readlib 'div'`.

The library provides basic arithmetic to calculate with fractions. To create a fraction, use **div.div** 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 multiplies 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 **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 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:

```
> readlib 'div'

> div.div(1, 2, 3) + div.div(1, 3):
2

> div.div(1, 2) * div(1, 3):

> div.div(1, 2) * div.div(1, 3):
div(1, 6)

> 2 * div.div(1, 3):
div(2, 3)

> div.todec(div.div(1, 2)):
0.5

> div.todiv(ans):
div(1, 2)       0
```

Relations:

Two fractions can be compared with the $<$, $<=$, $=$, $>=$, and $>$ operators.

Functions:

`div.denom (a)`

This function returns the denominator of the fraction `a` of the user-defined type `'div'` and returns it as a number.

The function is written in the Agena language and is included in the `lib/div.agn` file.

See also: **div.numer**.

div.div ([x,] y, z)

div.div ([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 the Agena language and is included in the `lib/div.agn` file.

div.numer (a)

This function returns the numerator of the fraction a of the user-defined type `'div'` and returns it as a number.

The function is written in the Agena language and is included in the `lib/div.agn` file.

See also: **div.denom**.

div.todec (a)

This function converts a fraction a of the user-defined type `'div'` to a float and returns it.

The function is written in the Agena language and is included in the `lib/div.agn` file.

See also: **div.todiv**.

div.todiv (x)

This function converts a number x to an improper fraction of the user-defined type `'div'` and returns it. The second return is the accuracy (see **math.fraction** for further information).

The function is written in the Agena language and is included in the `lib/div.agn` file.

See also: **div.todec**, **math.fraction**.

Chapter Eight

C API Functions

8 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[34]. For a description of the API functions taken from Lua, see its Lua 5.1 manual.

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 Endianess 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.

[34] 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.
xbase.h	Interface to dBASE III file support of the Shapelib library.

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_ccall

`agn_Complex agn_ccall (lua_State *L, int nargs, int nresults);` (Non-ANSI)

```
agn_Complex agn_ccall (lua_State *L, int nargs, int nresults,
   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 is 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 is issued. **agn_ccall** pops the function and its arguments from the stack.

agn_checkboolean

```
lua_Integer agn_checkboolean (lua_State *L, int idx);
```

Checks whether the value at index `idx` is the Boolean **true**, **false**, or **fail** and returns this Boolean as an integer, where 1 denotes **true**, 0 denotes **false**, and 2 denotes **fail**. An error is raised if the value at `idx` is not a Boolean.

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_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_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.

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_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 (ANSI version only)

```
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.

agn_complexreal (ANSI version only)

```
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.

agn_copy

```
LUA_API void agn_copy (lua_State *L, int idx)
```

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.

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.

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_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_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_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_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_free

```
void *agn_free (lua_State *L, ...);
```

De-allocates one or more blocks of memory pointed to by pointers of type void *. The last argument must be **NULL**.

See also: **agn_malloc**.

agn_getbitwise

```
void 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_getemptyline

```
void 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_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_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

```
void agn_getlibnamereset (lua_State *L)
```

Returns the current setting for the **restart** statement to also reset **libname** and pushes a Boolean on the stack.

See also: **agn_setlongtable**.

agn_getlongtable

```
void 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.

See also: **agn_setlongtable**.

agn_getnoroundoffs

```
void agn_getnoroundoffs (lua_State *L)
```

Returns the current mode used by for/in loops with step sizes that are not integral: 0 if the improved precision method to prevent round-off errors in iteration is not used, and 1 if it is.

See also: **agn_setnoroundoffs**.

agn_getrtable

```
LUA_API int agn_getrtable (lua_State *L, int idx)
```

Pushes the remember table if 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.

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_getutype

```
int agn_getutype (lua_State *L, int idx);
```

Returns the user-defined type of a procedure, table, sequence, set, 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_isfail

```
int agn_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**).

agn_isfalse

```
int agn_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**).

agn_isnumber

```
int agn_isnumber (lua_State *L, int idx);
```

Returns 1 if the value at the given acceptable index is a number, and 0 otherwise.

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

```
int agn_isstring (lua_State *L, int idx);
```

Returns 1 if the value at the given acceptable index 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

```
int agn_istrue (lua_State *L, int idx);
```

Returns 1 if the Boolean value at the given acceptable index results to **true**, 0 otherwise (**false** and **fail**).

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_ncall

```
lua_Number agn_ncall (lua_State *L, int nargs, int nresults);
```

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 is issued. **agn_ncall** pops the function and its arguments from the stack.

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_nops

```
size_t agn_nops (lua_State *L, int idx);
```

Determines the number of actual table, set, or sequence entries of the structure at stack index `idx`. If the value at `idx` is not a table, set, or sequence, 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_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_paircheckbooloption

```
agn_paircheckbooloption (lua_State *L, const char *procname, int idx,
   const char *option)
```

For the given Agena procedure `procname`, checks whether the value at index `idx` 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`.

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_pairgetnumbers

```
void agn_pairgetnumbers (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 `idx` is a pair. It then checks whether the left-hand and right-hand side are numbers and returns these numbers in `x` and `y`.

If the value at `idx` is not a pair, or if at least one of its operands is not a number, it issues an error.

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.

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_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 and the maximum number of items assignable to the sequence at index `idx` in `a`, a C array with two 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.

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_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_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_setnoroundoffs

```
void agn_setnoroundoffs (lua_State *L, int value)
```

Sets the mode used by **for/in** loops with step sizes that are not integral: pass 0 for `value` if the improved precision method to prevent round-off errors in iteration shall not used, and 1 if it shall be used.

See also: **agn_getnoroundoffs** .

agn_setreadlibbed

```
int agn_setreadlibbed (lua_State *L, const char *name)
```

Inserts name into the global set **package.readlibbed** .

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_setutype

```
void agn_setutype (lua_State *L, int idxobj, int idxtype);
```

Sets a user-defined type of a procedure, table, sequence, set, 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`.

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 two entries. The actual number of items is stored to a[0], the current allocable size to a[1].

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 five 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].

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_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_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 strings or names `'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 is 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_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 is 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 is 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_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** .

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.

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_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_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_pushundefined

```
void lua_pushundefined (lua_State *L);
```

Pushes the value **undefined** onto the stack.

lua_pushtrue

```
void lua_pushtrue (lua_State *L);
```

This macro pushes the Boolean value **true** onto the stack.

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_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 value 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 value 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 value 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_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_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_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.

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.

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_seqgetinumber

```
lua_Number lua_seqgetinumber (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.

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);
```

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.

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_seqrawget2

```
void lua_seqrawget2 (lua_State *L, int idx);
```

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.

Contrary to **lua_seqrawget**, the function does not issue an error if an index does not exist in the sequence. Instead, **null** is returned.

This function pops the key from the stack (putting the resulting value in its place). The function does not invoke any metamethods.

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 at 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);
```

This macro 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_sinsert

```
void lua_sinsert (lua_State *L, int idx);
```

This macro 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.

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_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

```
void lua_srawget (lua_State *L, int idx);
```

Checks whether the set at index `idx` contains the item at the top of the stack. The function pops this item from the stack putting the Boolean value **true** or **false** in its place.

This function pops the value from the stack. 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.

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.

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**.

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_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.

Appendices

Appendix A

A1 Operators

Unary operators are:

&&, ~~, ||, ^^, abs, arccos, arcsin, arctan, assigned, atendof, char, copy, cos, cosh, entier, even, exp, filled, finite, first, float, lngamma, gethigh, getlow, imag, instr, int, join, last, left, ln, lower, nargs, not, qsadd, real, replace, right, sadd, sign, sin, sinh, size, sqrt, tan, tanh, trim, type, unassigned, unique, upper, typeof, - (unary minus).

Binary operators are:

in, intersect, minus, shift, split, subset, union, xor, xsubset, + (addition), - (subtraction), * (multiplication), / (division), *% (percentage) /% (ratio), \ (integer division), % (modulus), ^ (exponentiation), ** (integer exponentiation), & (concatenation), = (equality), < (less than), <= (less or equal), > (greater than), >= (greater or equal), $ (substring), : (pair constructor), ! (complex constructor), && (bitwise and), || (bitwise or), ^^ (bitwise xor), ~~ (bitwise complement).

A2 Metamethods

The following metamethods were inherited from Lua 5.1:

Index to metatable	Meaning
'__index'	Procedure invoked when a value shall to be read from a table, set, sequence, or pair.
'__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
'__arctan'	**arctan** operator
'__cos'	**cos** operator
'__eeq'	strict equality operator (==)
'__entier'	**entier** operator
'__even'	**even** operator
'__exp'	**exp** operator
'__finite'	**finite** operator
'__lngamma'	**lngamma** operator
'__in'	**in** binary operator (for tables and sequences only)
'__int'	**int** operator
'__intdiv'	integer division
'__ipow'	exponentiation with an integer power
'__ln'	**ln** operator
'__qsadd'	**qsadd** operator for table or sequence based user-defined types
'__sadd'	**sadd** operator for table or sequence based user-defined types
'__sign'	**sign** operator
'__size'	**size** operator
'__sin'	**sin** operator
'__sqrt'	**sqrt** operator
'__tan'	**tan** operator
'__writeindex'	Procedure invoked when a value shall to be written to a table, set, sequence, or pair.

Table 21: Metamethods introduced with Agena

A3 System Variables

Agena lets you configure the following settings:

System variable	Meaning
environ.homedir	The path to the user's home directory
mainlibname	The path to the main Agena directory
libname	The paths to Agena libraries
environ.buffersize	The default buffer size for file operations for the **os.fcopy**, **net.receive**, and **binio.readlines** functions. It is equal to the C constant BUFSIZ in stdio.h. Grep LUAL_BUFFERSIZE in the C sources.
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 **environ.os**.
environ.gdidefaultoptions	A table with all default plotting options for some functions in the **gdi** package. This table is set by **gdi.setoptions** .
environ.libpatchlevel	The update version of the main Agena library (in lib/library.agn).
environ.maxlong	The maximum integral value of the C type int32_t on your platform; do not change this value. Grep LUAI_MAXINT32 in the C sources.
environ.maxnumber	The maximum value an Agena number can represent.
environ.maxpathlength	The maximum number of characters for a file path (excluding C's \0 character).
environ.minlong	The minimum integral value of the C type int32_t on your platform; do not change this value. Grep LUAI_MININT32 in the C sources.
environ.minnumber	The minimum value an Agena number can represent.
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.
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 **environ.cpu**.
environ.pathsep	The token that separates paths in libname; by default is ';'. Do not change this value as it is used by the **with** function. Grep LUA_PATHSEP in the C sources.

System variable	Meaning
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. Do not change **environ.release** .
environ.umaxlong	The maximum integral value of the C type unsigned long on 32+bit systems, and unsigned long on 16bit hardware; do not change this value. Grep LUAI_UINT32 in the C sources.
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.
environ.withverbose	If set to **false**, the **with** function will not display warnings, the initialisation string, and the short names assigned. Default is **true**.
_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)
_PROMPT	Defines the prompt Agena displays at the console. If unassigned, by default the prompt is `'> '`.
_RELEASE	Release information on the installed Agena release, returned as a string, e.g. 'AGENA >> 2.0.0'.

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 `library.agn` file.
See also:

- Chapter 7.20 for a description of the **kernel** functions for other settings.
- Appendix A5 for settings that control how Agena outputs data at the console.

A4 Command Line Usage

Agena can be used in the command line as follows:

```
agena [options] [script [arguments]]
```

This means that any option, an Agena script, 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:

A4.1 Using the `-e` Option

We may write a script with a text editor, e.g. one to print the sine of a number. This script may look like the following two lines:

```
n := n or Pi;  # if n is not set from the shell, just assign Pi to n
writeline(sin(n));
```

This script prints the sine to a user-given numeric argument which is passed by using the `-e` option and a string containing a valid Agena statement. It uses a variable `n` which you must assign via the `-e` option:

```
shell> agena -e "n := Pi/2" sin.agn
1
```

Note that you first have to enter the `-e` option along with the Agena statement, and then the name of the script.

A4.2 Using the internal args Table

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. Any options are accessible via negative keys. The name of the interpreter 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 it is run, the output is:

```
shell> agena args.agn 0
-1      agena
0       args.agn
1       0
```

Just play around with this a little bit.

Let us use our new knowledge: The script 'ln.agn' requires a string and a number and calculates the natural logarithm of this number. The number entered at the command line is entered into the **args** table as a string, so you first must convert it into a `real` number.

```
arg1 := args[1];
arg2 := tonumber(args[2]);

assume(arg1 :: string, 'expected a string');
assume(arg2 :: number, 'expected a number');
writeline(arg1, ln(arg2));
```

Use it:

```
shell> agena ln.agn "The natural logarithm of 1 is: " 1
The natural logarithm of 1 is: 0
```

A4.3 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 does not matter. The following shell statement resets the Agena prompt and starts the interpreter:

```
shell> agena -i -e "_PROMPT := 'AGENA> '"
AGENA>
```

A4.4 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. line 1) 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.

A4.5 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 `agena.ini`
-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.
-v	show version information and compilation time
--	stop handling options
-	execute stdin and stop handling options

Table 23: Command line options

A5 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 `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
environ.aux.printcomplex	function	defines how to print a complex value, overriding the built-in default

Table index	Type	Functionality
environ.printenclosestrings	string	if set, Agena outputs strings with the prepending and appending string assigned to environ.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)
```

A6 The Agena Initialisation File

You can customise your personal Agena environment via special initialisation files.

The initialisation files may include code written in the Agena language 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 and Haiku, 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 7	<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                             #
###########################################################################

readlib 'gdi';
gdi.setoptions(colour~'red', axescolour~'grey');
```

A7 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

Table 26: Escape sequences

A8 Backward Compatib ility

Aliases for deprecated functions in Agena versions prior to 1.0 are no longer automatically initialised at start-up. However, by entering

```
> readlib 'compat';
```

you can activate them in your current session if you prefer compatibility to Agena 1.0, and additionally

```
> readlib 'compat.0';
```

to restore a limited compatibility to Agena 0.x. 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 **readlib** the **mapm** package and then load the compat.agn file.

A9 Mathematical Constants

Constant	Meaning
degrees	Factor $1/\pi$*180 to convert radians to degrees
Eps	Equals 1.4901161193847656e-08
EulerGamma	Euler-Mascheroni constant, equals 0.57721566490153286061
E, Exp	Constant e = exp(1) = 2.71828182845904523536
I	Imaginary unit $\sqrt{-1}$
infinity	Infinity ∞
Pi	Constant π = 3.14159265358979323846
Pi2	Constant 2π = 6.28318530717958647692 6
PiO2	Constant $\pi/2$ = 1.57079632679489661923 2
PiO4	Constant $\pi/4$ = 0.78539816339744830961 6
radians	Factor $\pi/180$ to convert degrees to radians
undefined	An expression stating that it is undefined, e.g. a singularity
math.Phi	the Golden number $(1+\sqrt{5})/2$

A10 Some Few Technical Notes

All Solaris and Linux binaries of Agena have been crated with GCC 4.4.5.

All Windows binaries of Agena have been crated with MinGW/GCC 4.5.2.

All Mac OS X binaries of Agena have been crated with Apple's GCC 4.2.1.

The C Sources should be ANSI C99 compatible, mostly due to Agena's support of complex arithmetic. Since Agena 1.4.0, the sources have been successfully compiled with GCC in Solaris 10, Windows 2000 and above, Linux, Mac OS X 10.7, and DJGPP for the DOS version.

Appendix B

B1 MIT Licence

The Agena source files are distributed under the MIT licence reproduced below. This means that Agena is free software and can be used for both academic and commercial purposes at absolutely no cost.

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:

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License 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 License 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 License 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 Licenses 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 license 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 licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

GNU GENERAL PUBLIC LICENSE
TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License 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 License. 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 licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; 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 License and to the absence of any warranty; and give any other recipients of the Program a copy of this License 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 licensed as a whole at no charge to all third parties under the terms of this License.
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 License. (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 License, 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 License, 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 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 License.

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, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, 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 License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License 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 license 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 License.

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 License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license 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 License 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 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 License.

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 License 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 License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License 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 License 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 License, 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 License as published by
the Free Software Foundation; either version 2 of the License, 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 License for more details.

You should have received a copy of the GNU General Public License 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 License. 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 License 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 License instead of this License.

B3 Sun Microsystems Licence for the fdlibm IEEE 754 Style Arithmetic Library

B4 GNU Lesser General Public License

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

[This is the first released version of the Lesser GPL. It also counts as the successor of the GNU Library Public License, version 2, hence the version number 2.1.]

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public Licenses are intended to guarantee your freedom to share and change free software--to make sure the software is free for all its users.

This license, the Lesser General Public License, 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 license or the ordinary General Public License 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 Licenses 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 license, 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 license from a patent holder. Therefore, we insist that any patent license obtained for a version of the library must be consistent with the full freedom of use specified in this license.

Most GNU software, including some libraries, is covered by the ordinary GNU General Public License. This license, the GNU Lesser General Public License, applies to certain designated libraries, and is quite different from the ordinary General Public License. We use this license 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 License therefore permits such linking only if the entire combination fits its criteria of freedom. The Lesser General Public License permits more lax criteria for linking other code with the library.

We call this license the "Lesser" General Public License because it does Less to protect the user's freedom than the ordinary General Public License. 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 License 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 License.

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 License 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 License 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 License (also called "this License"). 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 License; 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 License and to the absence of any warranty; and distribute a copy of this License 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 License.
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 License, 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 License, 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 License.

3. You may opt to apply the terms of the ordinary GNU General Public License instead of this License to a given copy of the Library. To do this, you must alter all the notices that refer to this License, so that they refer to the ordinary GNU General Public License, version 2, instead of to this License. (If a newer version than version 2 of the ordinary GNU General Public License 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 License 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 License.

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 License. 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 License. You must supply a copy of this License. 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 License. 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 License, 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 License. Any attempt otherwise to copy, modify, sublicense, link with, or distribute the Library is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

9. You are not required to accept this License, 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 License. Therefore, by modifying or distributing the Library (or any work based on the Library), you indicate your acceptance of this License 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 License.

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 License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License 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 License 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 License.

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 License 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 License incorporates the limitation as if written in the body of this License.

13. The Free Software Foundation may publish revised and/or new versions of the Lesser General Public License 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 License 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 License).

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
modify it under the terms of the GNU Lesser General Public
License as published by the Free Software Foundation; either
version 2.1 of the License, or (at your option) any later version.

This library 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
Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public
License along with this library; if not, write to the Free Software
Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

Also add information on how to contact you by electronic and paper mail.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the library, if necessary. Here is a sample; alter the names:

Yoyodyne, Inc., hereby disclaims all copyright interest in the
library `Frob' (a library for tweaking knobs) written by James Random Hacker.

<signature of Ty Coon>, 1 April 1990
Ty Coon, President of Vice
That's all there is to it!

B5 SOFA Software Licence

Copyright (C) 2012
Standards Of Fundamental Astronomy Board
of the International Astronomical Union.

=====================
SOFA Software License
=====================

NOTICE TO USER:

BY USING THIS SOFTWARE YOU ACCEPT THE FOLLOWING SIX TERMS AND CONDITIONS WHICH APPLY TO ITS USE.

1. The Software is owned by the IAU SOFA Board ("SOFA").

2. Permission is granted to anyone to use the SOFA software for any purpose, including commercial applications, free of charge and without payment of royalties, subject to the conditions and restrictions listed below.

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.

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)

B7 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:

Appendix C

C1: Further Reading

A selection of books that helped a lot in the last two years when advancing Agena:

- Niklaus Wirth, Algorithmen und Datenstrukturen mit Modula-2,
- Roberto Ierusalimschy, Programming in Lua,
- Kurt Jung & Aaron Brown, Beginning Lua Programming,
- Jürgen Wolf, C von A bis Z,
- Federico Biancuzzi & Shane Warden (Ed.), Masterminds of Programming.

Index

A

B

C

D

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X

Z