



# A Programming Language

---

Primer and Reference  
for Version 0.33.2

by Alexander Walz  
June 26, 2010

AGENA Copyright 2006-2010 by Alexander Walz. All rights reserved.  
Portions Copyright 2006 Lua.org, PUC-Rio. All rights reserved.

See Appendix B for Licences.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this manual, and the author was aware of a trademark claim, the designations have been printed in initial caps or all caps.

**Contact:** In case you find bugs, errors in this manual, have proposals, or questions regarding Agena, please contact the author at: [adena.info@t-online.de](mailto:adena.info@t-online.de)

The latest release of Agena may be found at <http://adena.sourceforge.net>.

## Credits

### Chapter 7: Standard Library documentation

Large portions of Chapter 7 have been taken from the Lua 5.1 Reference Manual written by Roberto Ierusalimsky, Luiz Henrique de Figueiredo, Waldemar Celes. Used by kind permission.

#### **case of** statement

The original code was written by Andreas Falkenhahn and posted to the Lua mailing list on 01 Sep 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 12 September 2005.

#### **globals** base library function

The original Lua and C code for **globals** has been written by David Manura for Lua 5.1 in 2008 and published on [www.lua.org](http://www.lua.org). Because of crashes with library C functions passed to **globals**, the C source has been patched 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](http://lua.org).

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

taken from Eric Tetz's `k-m-number-suffix.patch` available at [lua.org](http://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'.

## calc.fsum

The modified Kahan algorithm used has been developed by Kazufumi Ozawa, published in his paper 'Analysis and Improvement of Kahan's Summation Algorithm'.

## calc.interp

taken from 'Numerical Algorithms with C' by Gisela Engeln-Müllges and Frank Uhlig.

## 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.

## **Advanced precision algorithm for for/to loops**

The method to prevent round-off errors in **for/to** loops 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.

## **Graphics**

The graphical capabilities of Agena in the UNIX 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 adaption of the Lua binding written by Luiz Henrique de Figueiredo, put to the public domain.

## **Year 2038 fix**

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

`arctan`, `exp2`, `gamma`, `lgamma`, `calc.dawson`, `calc.dilog`, `calc.Ci`, `calc.Chi`, `calc.Ei`, `calc.fresnelc`, `calc.fresnels`, `calc.Psi`, `calc.Si`, `calc.Shi`, and `calc.Ssi` functions

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](http://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.

## Table of Contents

<b>1 Introduction</b>	13
1.1 Abstract	13
1.2 Features	13
1.3 Features in Detail	14
1.4 History	15
<b>2 Installing and Running Agenda</b>	19
2.1 Solaris	19
2.2 Linux	19
2.3 Windows	20
2.4 OS/2 Warp 4 and eComStation	21
2.5 DOS	23
2.6 Mac OS X 10.5 and higher	23
2.7 Haiku	24
2.8 Agenda Initialisation	24
<b>3 Overview</b>	29
3.1 Input Conventions	29
3.2 Getting familiar	29
3.3 Useful Statements	30
3.4 Conditions	31
3.5 Loops	31
3.6 Procedures	33
3.7 Comments	33
<b>4 Data &amp; Operations</b>	37
4.1 Names, Keywords, and Tokens	37
4.2 Assignment	38
4.3 Enumeration	39
4.4 Deletion	40
4.5 Precedence	41
4.6 Arithmetic	41
4.6.1 Numbers	41
4.6.2 Arithmetic Operations	43
4.6.3 Increment and Decrement	45
4.6.4 Mathematical Constants	46
4.6.5 Complex Math	46
4.7 Strings	47
4.8 Boolean Expressions	52
4.9 Tables	54
4.9.1 Arrays	55
4.9.2 Dictionaries	59
4.9.3 Table, Set and Sequence Operators	60
4.9.4 Table Functions	62
4.9.5 Table References	63
4.10 Sets	64
4.11 Sequences	66
4.12 More on the create statement	70
4.13 Pairs	71

4.14 Other types .....	73
<b>5 Control</b> .....	77
5.1 Conditions .....	77
5.1.1 if Statement .....	77
5.1.2 is Operator .....	78
5.1.3 case Statement .....	79
5.2 Loops .....	79
5.2.1 while-Loops .....	79
5.2.2 for/to loops .....	80
5.2.3 for/in Loops over Tables .....	82
5.2.4 for/in Loops over Sequences .....	83
5.2.5 for/in Loops over Strings .....	83
5.2.6 for/in Loops over Sets .....	84
5.2.7 for/in Loops over Procedures .....	84
5.2.8 for/while Loops .....	85
5.2.9 Loop Interruption .....	85
<b>6 Programming</b> .....	89
6.1 Procedures .....	89
6.2 Local Variables .....	90
6.3 Global Variables .....	91
6.4 Changing Parameter Values .....	92
6.5 Optional Arguments .....	92
6.6 Passing Options in any Order .....	94
6.7 Type Checking & Error Handling .....	94
6.8 Multiple Returns .....	97
6.9 Shortcut Procedure Definition .....	97
6.10 User-Defined Procedure Types .....	98
6.11 Scoping Rules .....	98
6.12 Loops in Procedures .....	100
6.13 Packages .....	100
6.13.1 Writing a New Package .....	100
6.13.2 The with Function .....	101
6.14 Remember tables .....	103
6.14.1 Standard Remember Tables .....	103
6.14.2 Read-Only Remember Tables .....	105
6.14.3 Functions for Remember Tables .....	106
6.15 Overloading Operators with Metamethods .....	107
6.16 Extending built-in Functions .....	110
6.17 Closures: Procedures that Remember their State .....	112
6.18 File I/O .....	113
6.18.1 Reading Text Files .....	113
6.18.2 Writing Text Files .....	113
<b>7 Standard Libraries</b> .....	117
7.1 Basic Functions .....	117
7.2 Coroutine Manipulation .....	144
7.3 Modules .....	145
7.4 String Manipulation .....	146
7.4.1 Kernel Operators and Basic Library Functions .....	146



7.4.2 The strings Library .....	148
7.4.3 Patterns .....	156
7.5 Table Manipulation .....	159
7.5.1 Kernel Operators .....	159
7.5.2 tables Library .....	161
7.6 Set Manipulation .....	163
7.7 Sequence Manipulation .....	165
7.8 Mathematical Functions .....	168
7.8.1 Kernel Operators .....	168
7.8.2 Base Library Functions .....	171
7.8.3 math Library .....	176
7.9 Input and Output Facilities .....	179
7.10 binio - Binary File Package .....	187
7.11 Operating System Facilities .....	193
7.12 The Debug Library .....	203
7.13 utils - Utilities .....	206
7.14 stats - Statistics .....	209
7.15 calc - Calculus Package .....	210
7.16 linalg - Linear Algebra Package .....	217
7.17 clock - Clock Package .....	225
7.18 ads - Agena Database System .....	227
7.19 gdi - Graphic Device Interface package .....	236
7.19.1 Opening a File or Window .....	236
7.19.2 Plotting Functions .....	236
7.19.3 Colours, Part 1 .....	237
7.19.4 Closing a File or Window .....	237
7.19.5 Supported File Types .....	237
7.19.6 Plotting Graphs of univariate Functions .....	237
7.19.7 Plotting geometric Objects easily .....	238
7.19.8 Colours, Part 2 .....	239
7.19.9 GDI Functions .....	239
7.20 mapm - Arbitrary Precision Library .....	249
7.21 fractals - Library to Create Fractals .....	251
7.21.1 Escape-time Iteration Functions .....	251
7.21.2 Auxiliary Mathematical Functions .....	253
7.21.3 The Drawing Function fractals.draw .....	254
7.21.4 Examples .....	255
7.22 xbase - Library to Read and Write xBase Files .....	256
<b>8 C API Functions .....</b>	<b>265</b>
<b>Appendix A .....</b>	<b>289</b>
A1 Operators .....	289
A2 Metamethods .....	289
A3 System Variables .....	290
A4 Command Line Usage .....	292
A4.1 Using the -e Option .....	292
A4.2 Using the internal args Table .....	292
A4.3 Running a Script and then entering interactive Mode .....	293
A4.4 Running Scripts in UNIX and Mac OS X .....	293

---

A4.5 Command Line Switches .....	294
A5 Define your own Printing Rules for Types .....	294
A6 The Agenda Initialisation File .....	295
<b>Appendix B</b> .....	298
B1 MIT Licence .....	298
B2 GNU GPL v2 Licence .....	298
B3 Sun Microsystems Licence for the fdlibm IEEE 754 Style Arithmetic Library ....	305
B4 GNU Lesser General Public License .....	305
B5 Other Copyright remarks .....	315
B6 MAPM Copyright Remark (Mike's Arbitrary Precision Math Library) .....	316

## Chapter One

# Introduction



# 1 Introduction

## 1.1 Abstract

Agena is an easy-to-learn procedural programming language designed to be used in scientific, educational, linguistic, and many other applications.

Agena provides fast real and complex arithmetics, 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.

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://agena.sourceforge.net>

## 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.

The interpreter is based on the original Lua 5.1 sources created by Roberto Ierusalimsky, Luiz Henrique de Figueiredo, and Waldemar Celes.

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,
- 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 Features 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,
- 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 **try/else** statement and 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 'nothing',
- 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,
- an arbitrary precision mathematical library,
- 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,
- the **external** switch to a numeric **for** loop to pass the last iteration value 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,
- xBase file support.

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, was left almost 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.

## 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 to make the interpreter work in UNIX environments. The 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.



## Chapter Two

# Installing & Running Agenda



## 2 Installing and Running Agena

### 2.1 Solaris

In Solaris, 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/adena`. The `/usr/adena/lib` directory is called the `main Agena library folder`.

Make sure you have the *libgcc*, *ncurses*, and *readline* libraries installed. From the command line, type `adena` and press RETURN.

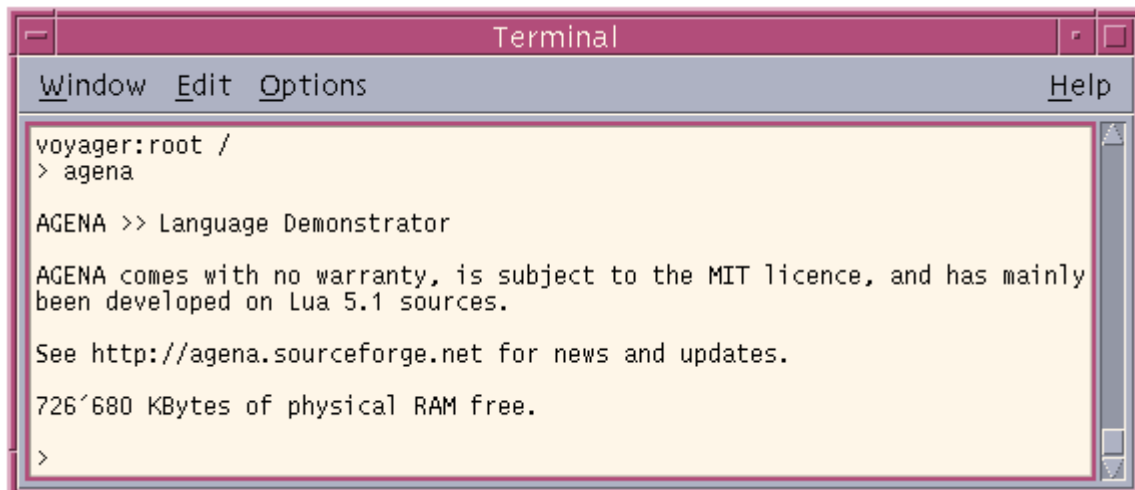


Image 1: Startup message in Solaris

The procedure for Solaris for x86 CPUs is the same. In Solaris, the package always installs as `SMCAgena`.

### 2.2 Linux

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

```
> sudo dpkg -i agena-x.y.z-linux-i386.deb
```

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

```
> rpm -ihv 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/adena`. The `/usr/adena/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.

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.

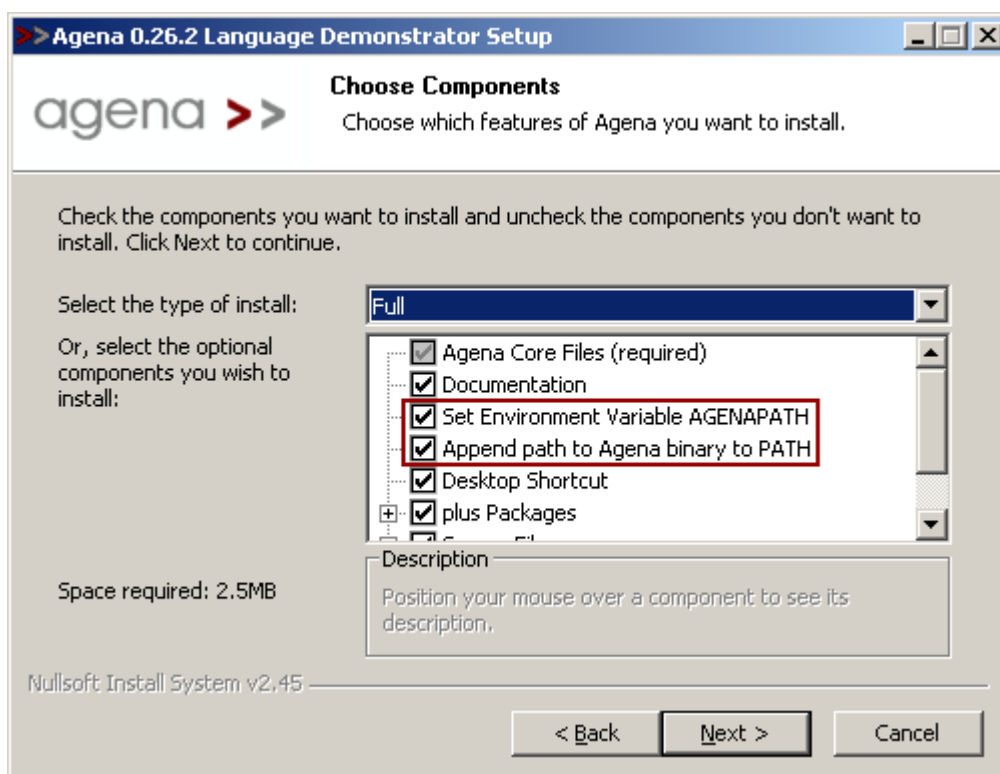


Image 2: Leave the framed settings checked

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

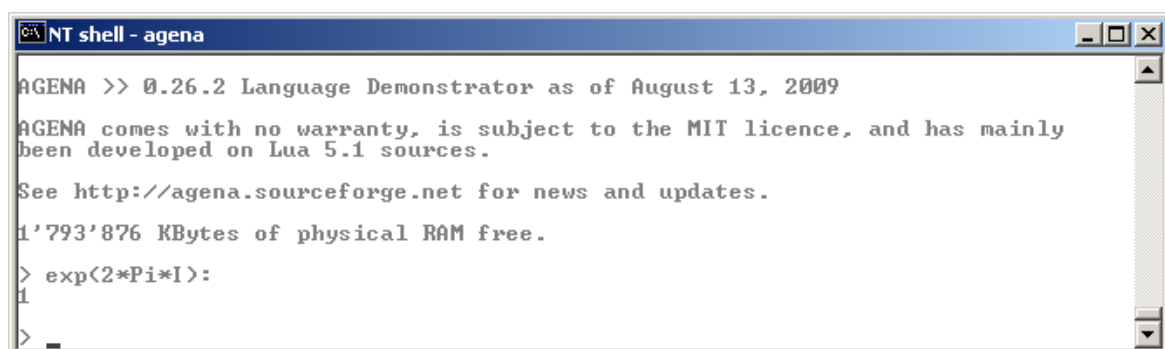


Image 3: Start-up message in Windows

## 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.

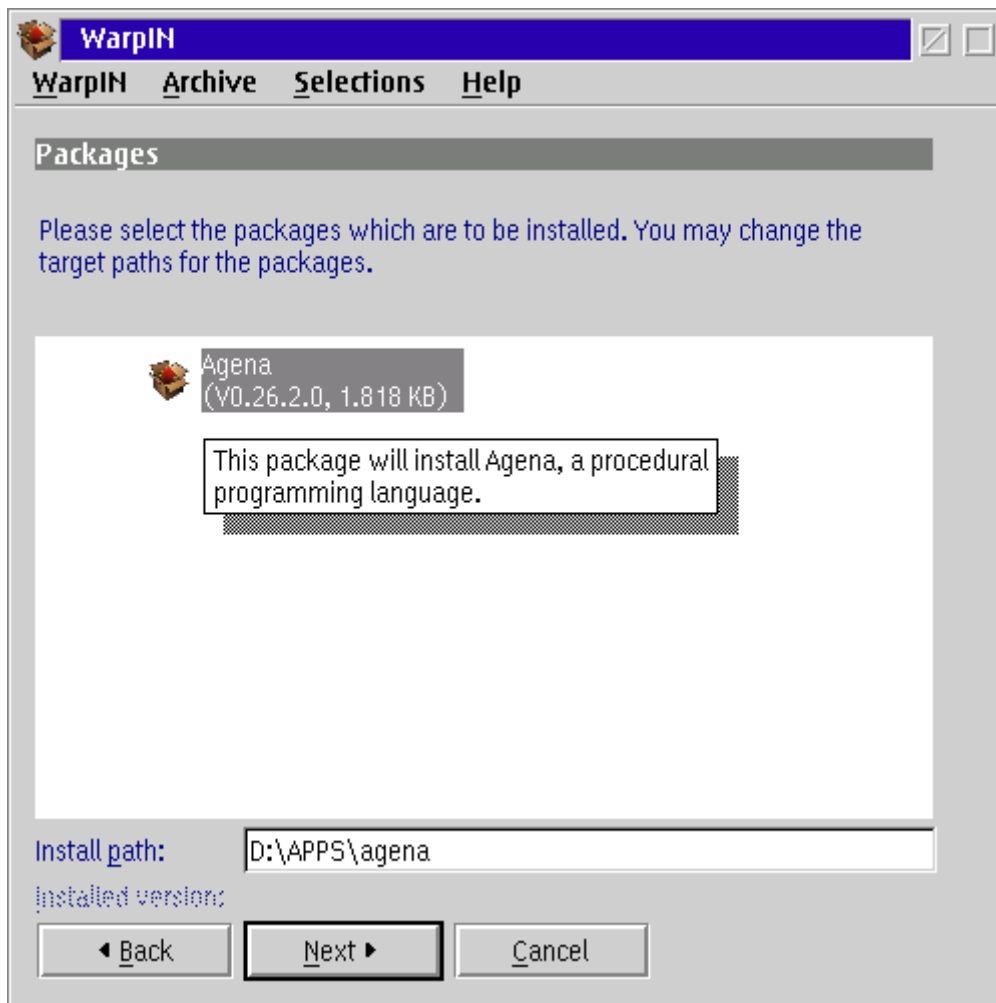


Image 4: Setting up Agena in OS/2 Warp 4

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) 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.

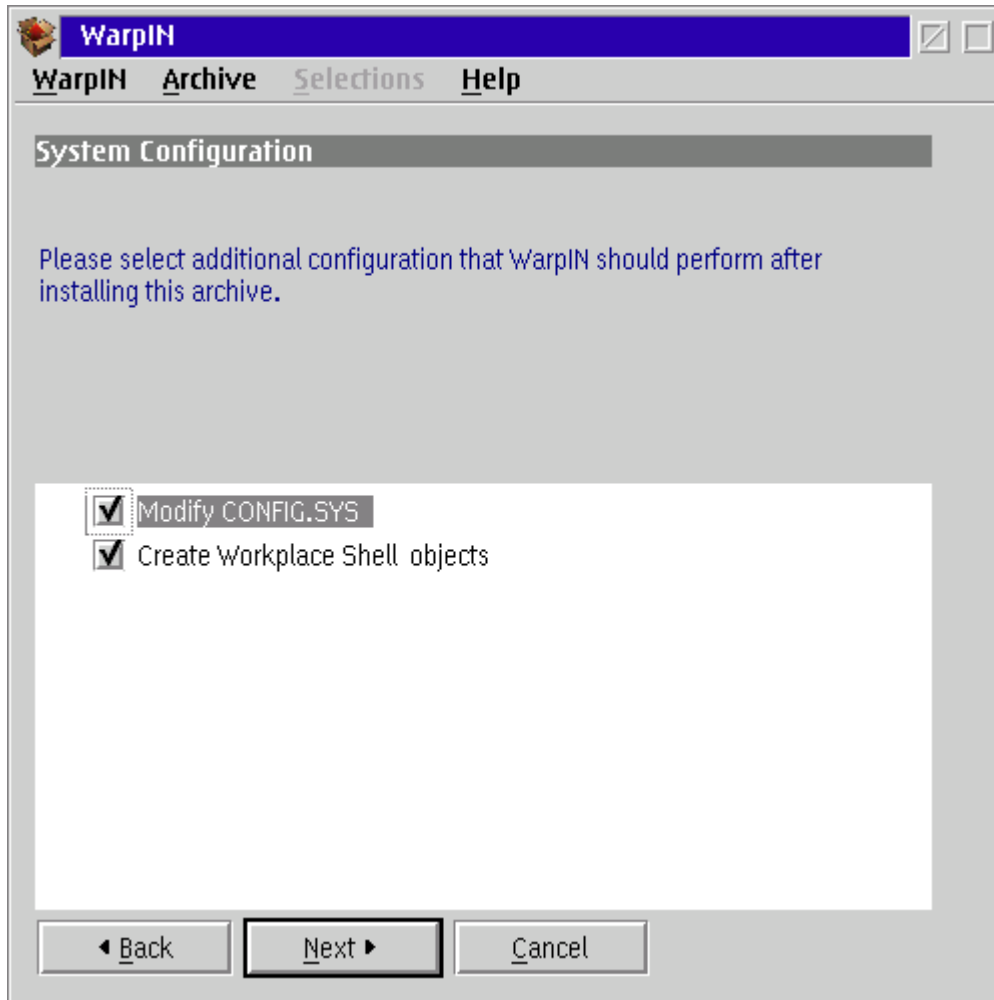


Image 5: Leave the settings checked

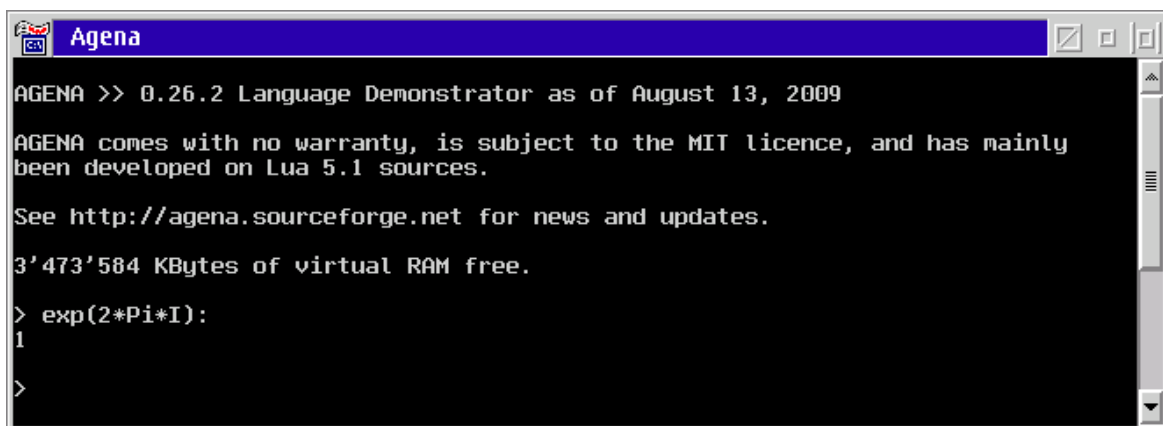


Image 6: Agena console in OS/2 Warp 4

## 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
```

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 in the **with** and **readlib** functions. If it could not be set, many 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 **homedir** environment variable.



## Chapter Three

### Overview



### 3 Overview

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.

#### 3.1 Input Conventions

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 Getting familiar

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

```
> 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:

```
> 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:

```
> arcsin(1):
1.5707963267949
```

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

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

Some of the most common functionality has been implemented with operators to ensure maximum speed. Try one of them. **lower** converts all letters from upper case to lower case.

```
> lower('AGENA'):  
agenda
```

One of the types to hold structured values is the *table*, which can hold any kind of data. Assume you would like to store the birthdays of your friends, enter:

```
> birthdays := ['Neo' ~ '1970/01/01', 'Trinity' ~ '1970/12/24'];
```

Determine Neo's birthday:

```
> birthdays['Neo']:  
1970/01/01
```

You can add new entries into your table.

```
> birthdays['Morpheus'] := '1952/04/01'
```

Now print its current content:

```
> birthdays:  
[Morpheus ~ 1952/04/01, Neo ~ 1970/01/01, Trinity ~ 1970/12/24]
```

To delete entries, just type:

```
> birthdays['Morpheus'] := null  
> birthdays:  
[Neo ~ 1970/01/01, Trinity ~ 1970/12/24]
```

### 3.3 Useful Statements

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

```
> ans:  
[Neo ~ 1970/01/01, Trinity ~ 1970/12/24]
```

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**:

```
> cls
```

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

```
> restart;
```

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

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

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

```
Agena$ restart;
```

### 3.4 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 := 'agenda';

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

### 3.5 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;

> c:
987
```

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

```
> c := 0;

> do
>   inc c
> as c < 10;

> c:
10
```

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

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

The **skip** statement 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
```

### 3.6 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;

> fact(10):
3628800
```

A procedure can call itself.

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

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

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

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

A function with two arguments:

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

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

### 3.7 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<sup>1</sup>.

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

Now let us learn more about Agena.

---

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

## 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 <b>undefined</b> and <b>infinity</b>
string	any text
boolean	booleans (e.g. <b>true</b> , <b>false</b> , and <b>fail</b> )
null	a value representing `nothing`
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 <b>null</b> 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.

### 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	lvar
_var	1_
var1	
_varln	
_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 band bnot bor bxor
break by bye case char clear cls copy cos cosh dec delete dict do elif
else end entier enum esac even exp external 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 trim true try type typeof unassigned union
unique upper while xor xsubset yrt
```

```
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
name1, name2, ..., namek := value1, value2, ..., valuek
name1, name2, ..., namek -> value
```

In the first form, one value is stored in one variable, whereas in the second form, called 'multiple assignment statement', name<sub>1</sub> is set to value<sub>1</sub>, name<sub>2</sub> is assigned value<sub>2</sub>, 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.

```
> 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
```

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 name1 [, name2, ... ]
enum name1 [, name2, ... ] from value
```

In the first form, *name<sub>1</sub>*, *name<sub>2</sub>*, 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

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

**clear**  $name_1$  [,  $name_2$ , ...,  $name_k$ ]

```
> 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.

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.



## 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`, or `D`, then the number is multiplied with 1,024, 1,048,576 (= 1,024<sup>2</sup>), 1,073,741,824 (= 1,024<sup>3</sup>), or 12, respectively. If a number ends in the letter `k` or `m`, then the number is multiplied with 1,000 or 1,000,000, 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	<code>0b&lt;binary number&gt;</code> or <code>0B&lt;binary number&gt;</code>	<code>0b10</code> = decimal 2
octal	<code>0o&lt;octal number&gt;</code> or <code>0O&lt;octal number&gt;</code>	<code>0b10</code> = decimal 9
hexadecimal	<code>0x&lt;hexadecimal number&gt;</code> or <code>0X&lt;hexadecimal number&gt;</code>	<code>0xa</code> = 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.

Beware of round-off errors in calculations involving very small or very large numbers. These will ultimately produce wrong results because Agena internally uses ANSI C numbers of double or complex double precision. The **mapm** package can be used in such situations because it provides arbitrary precision arithmetic. See Chapter 7.20 for more information.

### 4.6.2 Arithmetic Operations

Agena has the following arithmetical operators:

Operator	Operation	Details / Example
+	Addition	<code>1 + 2 » 3</code>
-	Subtraction	<code>3 - 2 » 1</code>
*	Multiplication	<code>2 * 3 » 6</code>
/	Division	<code>4 / 2 » 2</code>
^	Exponentiation with rational power	<code>2 ^ 3 » 8</code>
**	Exponentiation with integer power, faster than ^	<code>2 ** 3 » 8</code>
%	Modulus	<code>5 % 2 » 1</code>
\	Integer division	<code>5 \ 2 » 2</code>

Table 3: Arithmetic operators

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

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`<sup>2</sup> functions to activate the package before using these functions.

Unary operators<sup>3</sup> like **ln**, **exp**, etc. can be entered with or without simple brackets.

Procedure	Operation	Library	Example and result
<b>sin</b> (x)	Sine (x in radians)	Kernel	<code>sin(0) » 0</code>
<b>cos</b> (x)	Cosine (x in radians)	Kernel	<code>cos(0) » 1</code>
<b>tan</b> (x)	Tangent (x in radians)	Kernel	<code>tan(1) » 1.557407..</code>
<b>arcsin</b> (x)	Arc sine (x in radians)	Kernel	<code>arcsin(0) » 0</code>
<b>arccos</b> (x)	Arc cosine (x in radians)	Kernel	<code>arccos(0) » 1.570796....</code>
<b>arctan</b> (x)	Arc tangent (x in radians)	Kernel	<code>arctan(Pi) » 1.262627..</code>
<b>sinh</b> (x)	Hyperbolic sine	Kernel	<code>sinh(0) » 0</code>
<b>cosh</b> (x)	Hyperbolic cosine	Kernel	<code>cosh(0) » 1</code>
<b>tanh</b> (x)	Hyperbolic tangent	Kernel	<code>tanh(0) » 0</code>
<b>abs</b> (x)	Absolute value of x	Kernel	<code>abs(-1) » 1</code>
<b>entier</b> (x)	Rounds x downwards to the nearest integer	Kernel	<code>entier(2.9) » 2</code> <code>entier(-2.9) » -3</code>
<b>even</b> (x)	Checks whether x is even	Kernel	<code>even(2) » true</code>
<b>exp</b> (x)	Exponentiation $e^x$	Kernel	<code>exp(0) » 1</code>
<b>lngamma</b> (x)	$\ln \Gamma x$	Kernel	<code>exp(lngamma(3+1)) » 6</code>

<sup>2</sup> Check the **with** function which also provides an easy way to define short names for package procedures.

<sup>3</sup> See Appendix A1 for a list of all unary operators.

Procedure	Operation	Library	Example and result
<b>int(x)</b>	Rounds x to the nearest integer towards zero	Kernel	<code>int(2.9) » 2</code> <code>int(-2.9) » -2</code>
<b>ln(x)</b>	Natural logarithm	Kernel	<code>ln(1) » 0</code>
<b>log(x, b)</b>	Logarithm of x to the base b	Base	<code>log(8, 2) » 3</code>
<b>roundf(x, d)</b>	Rounds the real value x to the d-th digit	Base	<code>roundf(sqrt(2), 2) » 1.41</code>
<b>sign(x)</b>	Sign of x	Kernel	<code>sign(-1) » -1</code>
<b>sqrt(x)</b>	Square root of x	Kernel	<code>sqrt(2) » 1.414213..</code>
<b>sadd([...])</b>	Sum	Kernel	<code>sadd([1, 2, 3]) » 6</code>
<b>mean([...])</b>	Arithmetic mean	stats	<code>stats.mean([1, 2, 3]) » 2</code>
<b>median([...])</b>	Median	stats	<code>stats.median([1, 2, 3, 4]) » 2.5</code>

Table 4: Common mathematical functions

In addition, Agena can conduct bitwise operations on numbers.

Operator	Operation	Details / Example
<b>&amp;&amp;</b>	Bitwise `and` operation	<code>7 &amp;&amp; 2 » 2</code>
<b>  </b>	Bitwise `or` operation	<code>1    2 » 3</code>
<b>^^</b>	Bitwise `exclusive-or` operation	<code>7 ^^ 2 » 5</code>
<b>~~</b>	Bitwise complementary operation	<code>~~7 » -8</code>
<b>shift</b>	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 **kernel** function:

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

The default is restored as follows:

```
> 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<sup>4</sup> which are also around 10% faster:

<pre><b>inc</b> name [, value] <b>dec</b> name [, value]</pre>
--

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
```

---

<sup>4</sup> Finishing an **inc** or **dec** statement with a colon instead of a semicolon is refused.

#### 4.6.4 Mathematical Constants

Agena features the following arithmetic constants:

Constant	Meaning
<b>degrees</b>	Factor $1/\pi \cdot 180$ to convert radians to degrees
<b>Eps</b>	Equals $1.4901161193847656e-08$
<b>EulerGamma</b>	Euler-Mascheroni constant, equals $0.57721566490153286061$ .
<b>Exp</b>	Constant $e = \exp(1) = 2.71828182845904523536$
<b>I</b>	Imaginary unit
<b>infinity</b>	Infinity
<b>Pi</b>	Constant $\pi = 3.14159265358979323846$
<b>radians</b>	Factor $\pi/180$ to convert degrees to radians
<b>undefined</b>	An expression stating that it is undefined, e.g. a singularity

Table 6: Arithmetic constants

#### 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`, `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.

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.

By setting the environment variable `_EnvPrint.ZeroedCmplxVals` to **false**, Agena does *not* print complex values that are close to zero as just `0`. Internally, complex values are not rounded, however, when setting `_EnvPrint.ZeroedCmplxVals` to **true**.

## 4.7 Strings

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 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:

```
> '':
```

You may obtain a specific character by passing a dollar sign and its position in simple 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 with the **\$**-substring notation.

*string*\$( *start* [, *end*] )

You must at least pass the starting 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, 5):
rin

> str$(3, 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, -1):
ring

> str$(3, -2):
rin

> str$(-3, -2):
ng

> str$(-3):
i
```

If you want to retrieve only one single character from a string, you may also use the faster indexing method:

*string*[*pos*]

This returns the character in *string* that is at position *pos*. If you pass a negative number for *pos*, then the  $|pos|$ -th character from the right end of the string is returned.

```
> str := 'string'

> str[2]:
t

> str[-1]:
g
```

In Agena, a text can include any escape sequences 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.
```

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.

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

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

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<sup>5</sup>:

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

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

```
> `text:
text

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

Agena has basic operators useful for text processing:

Operator	Return	Function
<code>s in t</code>	number or <b>null</b>	Checks whether a substring <code>s</code> is included in string <code>t</code> . If true, the position of the first occurrence of <code>s</code> in <code>t</code> is returned; otherwise <b>null</b> is returned.
<code>s atendof t</code>	number or <b>null</b>	Checks whether a string <code>t</code> ends in a substring <code>s</code> . If true, the position of the position of <code>s</code> in <code>t</code> is returned; otherwise <b>null</b> is returned.

<sup>5</sup> For the current settings of your Agena version see the bottom of the `agnconf.h` file in the `src` directory of the distribution.

Operator	Return	Function
<b>replace</b> (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.
<b>s split d</b>	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.
<b>size</b> (s)	number	Returns the length of string s. If s is the empty string, 0 is returned.
<b>abs</b> (s)	number	Returns the numeric ASCII code of character s.
<b>char</b> (n)	string	Returns the character corresponding to the given numeric ASCII code n.
<b>lower</b> (s)	string	Converts a string to lowercase. Western European diacritics are recognised.
<b>upper</b> (s)	string	Converts a string to uppercase. Western European diacritics are recognised.
<b>trim</b> (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
```

Especially, the following functions can be used to find characters in a string:

Function	Functionality	Example
<b>in</b>	Returns the first position of a substring (left operand) in a string (right operand); if the substring cannot be found, <b>in</b> returns <b>null</b> . The operator is more than twice as fast as <b>strings.seek</b> .	'tr' in 'string' » 2
<b>instr</b>	Looks for the first match of a pattern (second argument) in a string (first argument). If it finds a match, then <b>instr</b> returns its position; otherwise, it returns <b>null</b> . A third, 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 <b>strings.find</b> .	instr('adena', '[aA]g', 1) » 1
<b>atendof</b>	Checks whether a string (second argument) ends in a substring (first argument). If true, the position of the position is returned; otherwise <b>null</b> is returned.	'ing' in 'raining' » 5
<b>strings.find</b>	Returns the first match of a substring (second argument) in a string (first argument) and returns the positions where the pattern starts <i>and ends</i> . An optional third argument specifies the position where to start the search. If it does not find a pattern, the function returns <b>null</b> .  The function supports pattern matching facilities described in Chapter 7.4.3.	strings.find('string', 'tr') » 2, 3  strings.find('string', 'tr', 3) » null  strings.find('string', 't.') » 2, 3

Function	Functionality	Example
<b>strings.seek</b>	<p>Returns the first match of a substring (second argument) in a string (first argument) and returns the position where the pattern starts. An optional third argument specifies the position where to start the search. If it does not find a pattern, the function returns <b>null</b>.</p> <p>Contrary to <b>strings.find</b>, the function does not support pattern matching and also does not return the ending position. The function is 30 % faster than <b>strings.find</b>.</p>	<pre>strings.seek(   'string', 'tr') » 2, 3  strings.seek(   'string', 'tr',   3) » null</pre>
<b>strings.rseek</b>	<p>Starting from the right end and always running to its left beginning, the function looks for the first match of a substring (second argument) in a string (first argument). The function returns the position where the pattern starts with respect to its left beginning. If it does not find a pattern, the function returns <b>null</b>.</p> <p>The function does not support pattern matching and also does not return the ending position.</p>	<pre>strings.rseek(   'string', 'ing') » 4  strings.rseek(   'string', 'ong') » null</pre>

Table 8: Search functions and operators

For more information on these functions, check Chapter 7.4.1 and Chapter 7.4.2. See also the descriptions of **strings.match** and **strings.gmatch**.

The replace operator can be used to find and replace characters in a string. See Table 9.

A string always is of type **string**.

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

## 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.

Name	Functionality	Example
<b>replace</b>	<p>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.</p> <p>A sequence of replacement pairs can be passed to the operator, too.</p> <p>You can also put a new string into a string if you pass the position of the character to be replaced.</p>	<pre>replace(str,   'string', 'text') » text</pre> <pre>replace('string',   seq('s':'S', 't':'T')) » STring</pre> <pre>strings.put('string',   2, 'T') » sTring</pre> <pre>strings.put('string',   2, 'TT') » sTTring</pre> <pre>strings.put('string',   2, '') » sring</pre>

Table 9: The **replace** operator

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

- relational operators (>, <, =, ==, <=, >=, <>),
- logical operators (**and**, **or**, **xor**, **not**),
- 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 equals for structures <sup>6</sup>	[1] == [1] 1 == 1
<>	not equals	1 <> 2

Table 10: Relational operators

<sup>6</sup> See Chapter 4.9.3.

Logical operators are:

Operator	Description	Examples
<b>and</b>	Both operands must evaluate to <b>true</b> so that the boolean expression results to <b>true</b> . Otherwise the result is <b>false</b> .	true and true » true false and false » false true and false » false false and true » false
<b>or</b>	At least one of the operands must evaluate to <b>true</b> so that the boolean expression results to <b>true</b> . If neither of the operands is <b>true</b> , the expression is <b>false</b> .	true or true » true true or false » true false or true » true false or false » false
<b>xor</b>	Returns <b>true</b> if exactly one of the operands evaluates to <b>true</b> , and the other one evaluates to <b>false</b> , <b>fail</b> , or <b>null</b> . It also returns <b>true</b> if exactly one of its operands evaluates to <b>null</b> and the other one is non- <b>null</b> .	true xor false » true true xor true » false false xor true » true 1 xor null » true
<b>not</b>	Turns a true expression to <b>false</b> and vice versa.	not true » false not false » true

Table 11: Logical operators

As expected, you can assign boolean expressions to names

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

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

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

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

The Boolean constant **fail** can be used to denote an error. With boolean operators (**and**, **or**, **not**), **fail** behaves like the **false** constant, 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 **null**.

## 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 `kernel('longtable')` is true (e.g. by entering the statement `kernel(longtable=true)`), then each key~value pair is printed at a separate line. If the setting `kernel('longtable')` is set to 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

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

$$[[value_1 [, value_2, \dots]]]$$

```
> 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:

<i>tablename[key]</i>
-----------------------

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

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

<i>tablename[key<sub>1</sub>][key<sub>2</sub>][...]</i> <i>tablename[key<sub>1</sub>, key<sub>2</sub>, ...]</i>
--

```
> 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
```



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 name1 [, table name2, ...]
name1 := [ ]
```

```
> 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:

```
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** 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.

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

**create table** *name<sub>1</sub>*(*size<sub>1</sub>*) [, **table** *name<sub>2</sub>*(*size<sub>2</sub>*), ...]

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:

$$[ [key_1 \sim value_1 [, key_2 \sim value_2, \dots]] ]$$

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

```
> B := [abs('p') ~ '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:

$$\text{create dict } name_1(size_1) [, \text{dict } name_2(size_2), \dots]$$

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

**create** {**table** | **dict**} *name*<sub>1</sub>[(*size*<sub>1</sub>)] [, {**table** | **dict**} *name*<sub>2</sub>[(*size*<sub>2</sub>)], ...]

#### 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
<b>c in A</b>	Boolean	Checks whether the structure A contains the given value c.
<b>filled A</b>	Boolean	Determines whether a structure contains at least one value. If so, it returns <b>true</b> , else <b>false</b> .
<b>A = B</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 <b>true</b> .
<b>A &lt;&gt; B</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 <b>false</b> .
<b>A == B</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 <b>true</b> .
<b>not(A == B)</b>	Boolean	The negation of A == B.
<b>A subset B</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 <b>true</b> if A = B.
<b>A xsubset B</b>	Boolean	Checks whether the values in structure A are also values in B. Contrary to <b>subset</b> , the operator returns <b>false</b> if A = B.
<b>A union B</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.
<b>A intersect B</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.
<b>A minus B</b>	table, set, seq	Returns all the values in A that are not in B as a new structure.

Name	Return	Function
<b>copy</b> A	table, set, seq	Creates a deep copy of the structure A, i.e. if A includes other tables, sets, or sequences, copies of these structures are built, too.
<b>join</b> A	string	Concatenates all strings in the table or sequence A.
<b>size</b> 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.
<b>sort</b> (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 <b>sort</b> is not an operator, so you must put the argument in brackets.
<b>unique</b> 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.
<b>sadd</b> A	number	Sums up all numeric table or sequence values. If the table or sequence is empty or contains no numeric values, <b>null</b> is returned. Sets are not supported.
<b>qsadd</b> 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, <b>null</b> 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
<b>put</b> (o, key, value)	Inserts a <i>key ~ value</i> pair into structure o.	It shifts up the original element at position <i>key</i> and all other elements.
<b>purge</b> (o, key)	Removes index <i>key</i> and its corresponding value from o.	All elements to the right are shifted down, so that no holes are created.

Table 13: Basic table procedures

Suppose we want to add a new entry 10 at position 3 of table C:

```
> 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]
```

For other functions, have a look into Chapter 7 of this manual and the Agenda 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 Agenda.

```
> A, B -> []
```

```
> A[1] := 1
```

```
> B:
[1]
```

## 4.10 Sets

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

$$\{ [ item_1 [, item_2, \dots] ] \}$$

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.

$$\text{insert } item_1 [, item_2, \dots] \text{ into } name$$

$$\text{delete } item_1 [, item_2, \dots] \text{ from } name$$

```
> 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<sub>1</sub>* [ (*size<sub>1</sub>*) ] [, **set** *name<sub>2</sub>* [ (*size<sub>2</sub>*) ], ...]

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.

The following operators work on sets:

Name	Return	Function
<code>c in A</code>	Boolean	Checks whether the set A contains the given value c.
<code>filled A</code>	Boolean	Determines whether a set contains at least one value. If so, it returns <b>true</b> , else <b>false</b> .
<code>A = B</code>	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 <b>true</b> .
<code>A &lt;&gt; B</code>	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 <b>false</b> .
<code>A == B</code>	Boolean	Same as =.
<code>A subset B</code>	Boolean	Checks whether the values in set A are also values in B. The operator also returns <b>true</b> if A = B.

Name	Return	Function
A <b>xsubset</b> B	Boolean	Checks whether the values in set A are also values in B. Contrary to <b>subset</b> , the operator returns <b>false</b> if $A = B$ .
A <b>union</b> 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 <b>intersect</b> B	set	Returns all values in two sets A, B that are included both in A and in B as a new set.
A <b>minus</b> B	set	Returns all the values in A that are not in B as a new set.
<b>copy</b> A	set	Creates a deep copy of the set A, i.e. if A includes other tables, sets, or sequences, copies of these structures are built, too.
<b>size</b> 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<sup>7</sup>. 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 indexed only with positive integers in the same fashion as table arrays are, starting at index 1. Other types of indexes are not allowed.

Suppose we want to define a sequence of two values. You may create it using the **seq** operator.

`seq( [ item1 [, item2, ...] ] )`

```
> a := seq(0, 1);
```

```
> a:
seq(0, 1)
```

You can access the items the usual way:

`seqname[numeric_key]`

<sup>7</sup> The structure was originally introduced to efficiently support objects like complex numbers or numeric ranges including a flexible way to pretty print them at the console.

```
> a[1]:
0

> a[2]:
1
```

If the index is larger than the current size of the sequence, an error is returned<sup>8</sup>.

```
> a[3]:
Error, line 1: index out of range
```

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.

```
> 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 program special operations only applicable to certain types of sequences.

A user-defined type can be deleted by passing **null** as a second argument to **settype**.

```
> settype(a, null);

> typeof(a):
sequence
```

The **create seq** 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 seq name1 [, seq name2, ...]
create seq name1(size1) [, seq name2(size2), ...]
```

---

<sup>8</sup> The error message can be avoided by defining an appropriate metamethod.

Items can be added only sequentially. You may use the **insert** statement for this or the conventional indexing method.

```
> create seq 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)
```

Sequences 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:

<b>pop bottom from</b> <i>name</i>
<b>pop top from</b> <i>name</i>

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 / sequence.

```
> stack := seq();

> insert 10, 11, 12 into stack;

> bottom(stack):
10

> top(stack):
12

> pop bottom from stack;

> pop top from stack;

> stack:
seq(11)
```

The 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
<b>insert</b>	Inserts one or more elements.	insert 1 into a
<b>delete</b>	Deletes one or more elements.	delete 0, 1 from a
<b>bottom</b>	Returns the item with key 1	bottom a
<b>top</b>	Returns the item with the largest key	top a
<b>copy</b>	Creates an exact copy of a sequence; deep copying is supported so that sequences inside sequences are properly treated.	b := copy a
<b>filled</b>	Checks whether a sequence has at least one item.	filled a
<b>getentry</b>	returns entries without issuing an error if a given index does not exist	getentry(a, 1, 3)
<b>in</b>	Checks whether an element is stored in the sequence, returns <b>true</b> or <b>false</b> .	0 in seq(1, 0)

Name	Description	Example
<b>join</b>	Concatenates all strings in a sequence in sequential order.	<code>join(a)</code>
<b>pop</b>	pops the first or the last element from a sequence	<code>pop left from a</code> <code>pop right from a</code>
<b>size</b>	Returns the current number of items.	<code>size a</code>
<b>sort</b>	Sorts a sequence in place.	<code>sort(a)</code>
<b>type</b>	Returns the general type of a sequence, i.e. <b>sequence</b> .	<code>type a</code>
<b>typeof</b>	Returns the user-defined type of a sequence, or the basic type if no special type has been defined.	<code>typeof a</code>
<b>unique</b>	Reduces multiple occurrences of an item in a sequence to just one.	<code>unique a</code>
<b>unpack</b>	Unpacks a sequence. See <b>unpack</b> in Chapter 7.1.	<code>unpack(a)</code>
<b>map</b>	Maps a function on all elements of a sequence.	<code>map(&lt;&lt; x -&gt; x^2 &gt;&gt;, seq(1, 2, 3));</code>
<b>zip</b>	Zips together either two sequences by applying a function to each of its respective elements.	<code>zip(&lt;&lt; x, y -&gt; x + y &gt;&gt;, seq(1, 2), seq(3, 4))</code>
<b>intersect</b>	Searches all values in one sequence that are also values in another sequence and returns them in a new sequence	<code>seq(1, 2)</code> <code>intersect</code> <code>seq(2, 3)</code>
<b>minus</b>	Searches all values in one sequence that are not values in another sequence and returns them as a new sequence.	<code>seq(1, 2)</code> <code>minus seq(2, 3)</code>
<b>subset</b>	Checks whether all values in a sequence are included in another sequence.	<code>seq(1)</code> <code>subset seq(1, 2)</code>
<b>union</b>	Concatenates two sequences simply by copying all its elements.	<code>seq(1, 2)</code> <code>union seq(2, 3)</code>
<b>settype</b>	Sets a user-defined type for a sequence.	<code>settype(a, 'duo')</code>
<b>gettype</b>	Returns a user-defined type for a sequence.	<code>gettype(a)</code>
<b>setmetatable</b>	Assigns a metatable to a sequence.	<code>setmetatable</code> <code>(a, mtbl)</code>
<b>getmetatable</b>	Returns the metatable stored to a sequence.	<code>getmetatable(a)</code>

Table 15: Basic sequence procedures

For more functions, consult the *Agena Quick reference Excel sheet*.

#### 4.12 More on the `create` statement

You cannot only initialise any number of tables 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()      []
```

### 4.13 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:

<b>left</b> [( <i>pair</i> )] <b>right</b> [( <i>pair</i> )]
---

```
> 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;
```

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
```

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
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 <b>true</b> or <b>false</b> . If at least one value x, a, b is not a number, the operator returns <b>fail</b> .	1.5 in 1:2
left	Returns the left operand of a pair.	left(a)
right	Returns the right operand of a pair.	right(a)
type	With pairs, always returns 'pair'.	type(a)
typeof	Returns either the user-defined type of the pair, or the basic type ('pair') if no special type was defined for the pair.	typeof(a)
settype	Sets a user-defined type for a pair.	settype(a, 'duo')
gettype	Returns the user-defined type of a pair.	gettype(a)
setmetatable	Sets a metatable to a pair.	setmetatable(p, mtbl)
getmetatable	Returns the metatable stored to a pair.	getmetatable(p)

Table 16: Operators and functions applicable to pairs



#### 4.14 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 condition1 then
    statements1
[elif condition2 then
    statements2]
[else
    statements3]
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
    
```

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
    expression1
else
    expression2
si
```

This means that the result is *expression*<sub>1</sub> if *condition* is **true** or any other value except **false**, **fail**, or **null**; and *expression*<sub>2</sub> 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 value11 [, value12] then statements1
  [of value21 [, value22] then statements2]
  [of ...]
  [else statementsk]
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
```

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('this will never be 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
```

**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 [external] 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  $start \leq stop$ . If so, it executes *statements* and returns to the top of the loop, increments *name* by *step* and then checks whether the new value is less or equal *stop*. If so, *statements* are executed again. 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 loop control variable is local to the loop body, so it cannot be used after looping completed. However, if you put the **external** keyword in front of the control variable, you will have access to the control variable after looping completed and may use its value in subsequent statements. This rule applies only to for/from/to-loops with or without a **while** extension. Note that if you use the **external** option within procedures, you usually want to declare the loop control variable as local, otherwise it will be treated as a global variable.

```
> for external i to 1e300 while fact(i) < 1k do od
> i:
7
```

When using the **external** switch 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<sup>9</sup> 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.

---

<sup>9</sup> See chapter 5.2.8 for more information in the **break** statement.

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, then Agena does not execute the loop and continues with the statement following it.

Let us first concentrate on table iteration.

```
for key, value in tbl do
  statements
od
```

The loop iterates over all key~value pairs in table *tbl* and with each iteration assigns the respective key to *key*, and its value to *value*.

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

> for i, j in a do
>   print(i, j)
> od
1      4
2      5
3      6
```

There are two variations: When putting the 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, the **external** switch is not supported. 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.

## 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 except the **external** option mentioned in the previous subchapter 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  $n \geq 0$  then the procedure returns null which quits the for/in iteration. See Chapter 6 which describes procedures in detail.

```
> double := proc(state, n)
>   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 [external] 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 as 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( [par1 [::type1] [, par2 [::type2], ...] ) [is]
    [local name1 [, name2, ...]];
    statements
end
```

All the values that a procedure shall process are given as *parameters* *par*<sub>1</sub>, 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). The **is** keyword is optional.

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.

```
return value [, value2, ...]
```

As you can see, you may not only return a single result, but also multiple ones.

Also, a procedure might not necessarily return anything - in this case do not use the **return** statement at all. If no **return** statement is given, the procedure does not even return the **null** value.

The following procedure computes the factorial of an integer<sup>10</sup>:

```
> 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:

$$\text{funcname}([arg_1 [, arg_2, \dots]])$$

```
> 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 lead to 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 and assign values later to this variable, then it is global. Note that control variables in **for** loops are always implicitly declared local if the **external** switch is not used, so we do not need to explicitly declare them.

Local declarations come in different flavours:

```
local name1 [, name2, ...]
local name1 [, name2, ...] := value1 [, value2, ...]
local name1 [, name2, ...] -> value
local enum name1 [, name2, ...] [from value]
```

In the first form,  $name_1$ , etc. are declared local.

---

<sup>10</sup>The library function **fact** is much faster.

In the second and third form, *name<sub>1</sub>*, etc. are declared local followed by initial assignments of values to these names.

In the last form, *name<sub>1</sub>*, etc. are declared local with a subsequent enumeration of those 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 result = infinity then break fi
>   od;
>   return result
> end;

> fact(10):
3628800
```

result has been declared local so it has no value at the interactive level.

```
> result:
null
```

### 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` a value 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 optionally 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<sup>11</sup>.

---

<sup>11</sup> If you need to conduct a Boolean equality operation (= operator) in a function call, such like `f(a=b)`, use the **isEqual** function, like `f(isEqual(a, b))`.

```
> 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
```

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
```

## 6.7 Type Checking & Error Handling

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 and user-defined type.
3. 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.
4. The **try** statement checks whether one or more values are of a basic type.

The language also provides the **error** handling function that interrupts the execution of a procedure and prints an error message if given.

The following types are available in Agena:

```
boolean, complex, lightuserdata, null, number, pair, procedure,
sequence, set, string, table, thread, userdata.
```

These names are reserved keywords, but 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(1):
number

> fact := proc(n) is
>   if type(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 also optionally specify 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).
```

Another efficient way of type checking is provided by the **try** statement.

```
try name1 [, name2, ...] :: typename1, [name3 [, name4, ...] :: typename2, ...]
```

```
try name1 [, name2, ...] :: typename1 else errorstring1  
[, name3 [, name4, ...] :: typename2 else errorstring2, ...]
```

In the first form, a standard error message is displayed and further computation stops. In the second form, a user defined error text is printed and execution of the function is interrupted.

```
> fact := proc(n) is
>   try n :: number;
>   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, at line 2:
  expected number, got string for local `n`

> fact := proc(n) is
>   try n :: number else 'bad value for argument';
>   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, at line 2:
  for local `n`: bad value for argument
```

Note that the **type** operator and the **try** statement only check for basic types. If you want to check user-defined types for procedures, tables, sequences, sets, and pairs, you should use the double colon notation or the **typeof** operator.

You can specify a user-defined type in the parameter list. Suppose you have defined a type called `triple`:

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

> settype(t, 'triple')

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

> sum(t):
6
```



## 6.8 Multiple Returns

As stated before, a procedure can return no, one, or more values. There are two ways to use these multiple returns in subsequent statements.

Consider the **strings.find** library function. It searches for a pattern in a string and returns the first and the final position of the pattern as two numbers.

```
> strings.find('Wulfila', 'ila'):
5          7
```

If you assign the return to only one variable, e.g.

```
> m := strings.find('Wulfila', 'ila'):
5
```

the second return is lost, so enter:

```
> m, n := strings.find('Wulfila', 'ila');

> m:
5

> n:
7
```

A function may also return a variable number of values. To store any of these returns for later access, just put the returns in a sequence or table:

```
> seq(strings.find('Wulfila', 'ila')):
seq(5, 7)
```

## 6.9 Shortcut Procedure Definition

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

```
<< [(] [par1 [:: type1] [, par2 [:: type2], ...]] [)] -> 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 the 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.

### 6.10 User-Defined Procedure Types

The **settype** function allows to group procedures  $\text{proc}_1, \text{proc}_2, \dots$ , by giving them a specific type (passed as a string) just as it does with sequences, tables, sets, and pairs.

**settype**( $\text{proc}_1$  [,  $\text{proc}_2, \dots$ ], 'your\_proctype')

The **typeof** operator returns the user-defined type of an object as a string. If no special type has been defined, it returns its basic type. The latter also applies to data types where **settype** cannot set user-defined types.

**typeof**( $\text{proc}_1$ )

The **type** operator does not return the user-defined type even if it is set, it will always return the basic type of an object.

```
> f := << x -> 1 >>

> settype(f, 'constant')

> typeof(f):
constant

> type(f):
procedure
```

### 6.11 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.

Variables declared local within procedures are only visible in these procedures.

Variables declared local in the **then** clauses of an **if**-statement live only in the respective **then** part. The same applies to variables declared local 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 local in **for**- or **while**-loops are only accessible in the bodies of these loops. The loop control variables of **for/to**- and **for/in**-loops are implicitly declared local to the respective loop bodies, with the exception of the **external** facility of **for/to** loops which is described in the next subchapter.

```
> 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
```

## 6.12 Loops in Procedures

As already noted, the control variable of a for/to loop is only local to the loop itself - but if you use the **external** keyword in the loop declaration, you will have access to it after execution of the loop completed. Make sure that in this case, you define the control variable local.

```
> mandelbrot := proc(x, y, iter, radius) is
>   local i, c, z;
>   z := x!y;
>   c := z;
>   for external 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 the number of iterations a complex value  $z$  takes to escape a given radius by applying it to the formula  $z = z^2 + c$ . Since the loop control variable  $i$  has been declared external, it can be used in the **return** statement.

```
> 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 external i to 3 do
>     local j;
>     for external j to 3 do od;
>     print(i, j)
>   od;
>   print(i, j)
> end;

> f()
1      4
2      4
3      4
3      null
```

## 6.13 Packages

### 6.13.1 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 `isInteger` function to this table.

```
> create table helpers;

> helpers.isInteger := << x -> int(x) = x >>; # aux function

> helpers.ndigits := proc(n::number) is
>   if not helpers.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):
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.

```
> restart;

> run 'd:/agena/helpers/helpers.agn'

> helpers.ndigits(10):
2
```

You may print the contents of the package table at any time:

```
> helpers:
[isInteger ~ procedure(0044A6E0), ndigits ~ procedure(0044A850)]
```

### 6.13.2 The with Function

The **with** function besides loading the package in a convenient way, automatically assigns short names to all or a user-defined set of 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 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 `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** is not reset by the **restart** statement because **restart** does not touch the contents of this specific system variable.

```
> with 'helpers'
isInteger, ndigits
```

```
> isInteger(1); # same as helpers.isInteger(1)
```

You may also want **with** to print a start-up notice at every package invocation if you assign 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 December 24, 2007\n';
```

```
> restart;
```

```
> with 'helpers'
helpers v1.0 as of December 24, 2007
```

```
isInteger, ndigits
```

Since you may not want that short names are set for auxiliary functions, you can put the names of all procedures for which short names shall be assigned as strings into the `packagename.loaded` table using the **register** function. Insert the following line to your `helpers.agn` file at its end:

```
register(helpers, 'ndigits');
```

The contents of the `helpers.agn` file should finally look like this:

```
create table helpers;
```

```
helpers.initstring := 'helpers v1.0 as of December 24, 2007\n';
```

```
helpers.isInteger := << x -> int(x) = x >>; # aux function
```

```
helpers.ndigits := proc(n:number) is
```

```

    if not helpers.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;

register(helpers, 'ndigits');
```

Save the file again and restart Agena.

```

> restart;

> with 'helpers'
helpers v1.0 as of December 24, 2007

ndigits
```

If your package includes an initialisation routine, then it will be run after the package has been found successfully. The name of the initialisation routine must be of the form ``packagename.init``, e.g.:

```

> helpers.init := proc() is
>     writeline('I am run')
> end;
```

## 6.14 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.

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.

### 6.14.1 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 **rinit** function and optionally filled with default values with the **rset** function. Since those functions are very basic, a more convenient facility is the **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  $f(0) = \text{undefined}$ .

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 **remember** function can be used to initialise rtables, explicitly set predefined values to them, and add further values later in a session.

```
> 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, the 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 7.1 for more details on their use.

### 6.14.2 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 `rset` function (or a utility that is based on `rset`). This gives you total control of 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 nonnegative ?
>     return exp(lngamma(x+1))
>   else
>     return undefined
>   fi
> end;
```

The **defaults** function can set up the rotatable 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 rotatable than if there would be no such value cache, because it would have to 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 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**.

### 6.14.3 Functions for Remember Tables

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

Procedure	Details
<b>hasrtable</b> (f)	Checks whether procedure $f$ possesses an rtable.
<b>rget</b> (f)	Returns the rtable of function $f$ .
<b>rinit</b> (f)	Initialises a standard remember table for the function $f$ .
<b>roinit</b> (f)	Initialises a read-only remember table for the function $f$ .
<b>rset</b> (f, argument, return) <b>rset</b> (f, [arguments], [returns])	Adds function argument(s) and the corresponding return(s) to the rtable of procedure $f$ .
<b>rdelete</b> (f)	Deletes the rtable of function $f$ entirely. If you want to use a new rtable with the function, you have to initialise it with <b>rinit</b> again.
<b>rwrite mode</b> (f)	Returns true if a function has a standard remember table, false if it has a read-only remember table, and fail if it has no remember table at all.

Table 17: Functions for remember tables

## 6.15 Overloading Operators with Metamethods

One of the many useful functions inherited from Lua 5.1 are metamethods which provide a means to apply 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<sup>12</sup>.

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 seq 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 seq 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
```

---

<sup>12</sup>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 the 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 seq 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  $x + yi$  notation. This can be done with the `\_\_tostring` method which must return a string.

```
> cmplx_mt.__tostring := proc(z) is
>   return if z[2]<0 then z[1]&z[2]&'i' else z[1]&'+ '&z[2]&'i' 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  $i$  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
```

## 6.16 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  $f(x) = \text{undefined}$ . 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 `agena.ini` file, located either in the `lib` folder of your Agena installation, or your home directory. Since files have their own `scope`, the **scope** and **epocs** keywords are no longer needed (but can be left in the file).

### 6.17 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
```

## 6.18 File I/O

Agenda features various functions to deal with files, to read lines and write values to them. Most of the functions come from Lua. All the functions processing files are included in the **io** and the **binio** packages.

### 6.18.1 Reading Text Files

One of the most useful functions to read in a text file line by line is the **io.lines** procedure which accepts the name of the file to be read as a string. They are usually used in **for** loops. The line read is stored to the loop key, the loop value is always **null**.

```
> for i, j in io.lines('d:/agenda/lib/agenda.ini') do
>   print(i, j)
> od
execute := os.execute;      null
getmeta  := getmetatable;   null
setmeta  := setmetatable;   null
```

### 6.18.2 Writing Text Files

To write numbers or strings into a file, we must first create it with the **io.open** function. The second argument tells Agenda to open the file in ``write`` mode.

```
> file := io.open('d:/file.text', 'w');
```

**io.open** returns an integer, a so-called file handle. File handles are used in many IO functions, e.g. the **write** procedure.

```
> io.write(file, 'I am a text.');
```

```
> io.write(file, 'Me ', 'too.');
```

After all values have been written, the file must be closed with **io.close**.

```
> io.close(file);
```

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 accessed and stored 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 to the file. The keys and values are separated by a pipe `'|'`, and a newline is inserted after each key~value pair has been added. 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);
```

## Chapter Seven

# Standard Libraries



## 7 Standard Libraries

The standard libraries taken from the Lua 5.1 distribution 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.

#### **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.

#### **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.

#### **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".

#### **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\_hashholes' stating whether the array part contains at least one hole.

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'.

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'.

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.

#### **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:  $\text{Beta}(x, y) = \frac{\Gamma x \Gamma y}{\Gamma(x+y)}$ , 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.

#### **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.

**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.

**bottom** (*obj*)

With the sequence *obj*, the operator returns the element at index 1. If the sequence is empty, it returns **null**.

See also: **top**.

**bye**

Quits the Agena session. No arguments or brackets are needed.

**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.

**concat** (*obj* [, *sep* [, *i* [, *j*]])

Returns *obj*[*i*] & *sep* & *obj*[*i*+1] ... *sep* & *obj*[*j*], where *obj* is either a table or sequence of strings. 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. If *i* is greater than *j*, returns the empty string. The empty string is also returned, if *obj* consists entirely of non-strings.

Use the **toString** function if you want to concatenate other values than strings, e.g.:

```
> concat(map(toString, [1, 2, 3])):
123
```

**countitems** (*item*, *s*)

**countitems** (*f*, *s* [, ...])

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

In the second form, by passing a function *f* with a Boolean relation as the first argument, all elements in *s* that satisfy the given relation are counted.



The return is a number. The function may invoke metamethods.

See also: **select**.

```
defaults (f)
defaults (f, tab)
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 **remember** function for further details.

```
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**.

```
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. Function **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.

```
_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 **selfenv** 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**.

**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*.

**getentry (o [, k<sub>1</sub>, ..., k<sub>n</sub>])**

Returns the entry `o[k1, ..., kn]` from the table or sequence `o` without issuing an error if one of the given indices (second to last argument) does not exist. It conducts a raw access and thus does not invoke any metamethods.

If `o[k1, ..., kn]` does not exist, **null** is returned. If only `o` is given, it is simply returned.

**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.

**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`.

### **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**.

### **globals (f)**

Determines<sup>13</sup> whether function `f` includes global variables (names which have not been defined local).

### **has (s, x)**

Checks whether the structure `s` (a table, set, sequence, or pair) contains element `x`. With tables, both indices (keys) and entries are scanned (if the index is a set, table, pair, or sequence, the index is not scanned, however). 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 is written in the Agenda language and included in the `library.agn` file.

### **hasrotable (f)**

Checks whether function `f` has a read-only remember table (that cannot be updated by the **return** statement). It returns **true** if it has got one, and **false** otherwise.

### **hasrtable (f)**

Checks whether function `f` has a remember table (that can be updated by the **return** statement). It returns **true** if it has got one, and **false** otherwise.

### **isBoolean (obj)**

Checks whether `obj` is of type **boolean** and returns **true** or **false**.

### **isComplex (obj)**

Checks whether `obj` is of type **complex** and returns **true** or **false**.

---

<sup>13</sup>Note that the function not always returns all global names.

**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.

**isNegint (x)**

Checks whether the number `x` is a negative integer and returns **true** or **false**. If `x` is not a number, the function returns fail.

**isNonnegint (x)**

Checks whether the number `x` is 0 or a positive integer and returns **true** or **false**. If `x` is not a number, the function returns fail.

**isNumber (obj)**

Checks whether `obj` is of type **number** and returns **true** or **false**.

**isNumeric (obj)**

Checks whether `obj` is of type **number** or of type **complex** and returns **true** or **false**.

**isPair (obj)**

Checks whether `obj` is of type **pair** and returns **true** or **false**.

**isPosint (x)**

Checks whether the number `x` is a positive integer and returns **true** or **false**. If `x` is not a number, the function returns fail.

**isselfref (s)**

Checks whether a structure `s` (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.

**isSequence (obj)**

Checks whether `obj` is of type **sequence** and returns **true** or **false**.

**isString (obj)**

Checks whether `obj` is of type **string** and returns **true** or **false**.

**isStructure (obj)**

Checks whether `obj` is of type **table**, **set**, **sequence**, or **pair** and returns **true** or **false**.

**isTable (obj)**

Checks whether `obj` is of type **table** and returns **true** or **false**.

**kernel (setting)**

**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'	<b>true</b> or <b>false</b>	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'	<b>true</b> or <b>false</b>	If set <b>true</b> (the default), two input prompts are always separated by an empty line. If set <b>false</b> , no empty line is inserted.
'libnamereset'	<b>true</b> or <b>false</b>	If set <b>true</b> , the <b>restart</b> statement resets <b>libname</b> and <b>mainlibname</b> to their original values. Default is <b>false</b> .
'longtable'	<b>true</b> or <b>false</b>	If set <b>true</b> , 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 <b>false</b> .
'signedbits'	<b>true</b> or <b>false</b>	If set to <b>true</b> , the bitwise operators <b>&amp;&amp;</b> , <b>~~</b> , <b>  </b> , <b>^^</b> , and <b>shift</b> internally use signed integers (the default), otherwise they use unsigned integers.

Examples:

```
> kernel('signedbits'):
true
```

```
> kernel(signedbits = false):
false
```

**left (obj)**

With the pair `obj`, the operator returns its left operand.

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.

**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.

**loadfile ([filename])**

Similar to **load**, but gets the chunk from file `filename` or from standard input, if no file name is given.

**loadstring (string [, chunkname])**

Similar to **load**, but gets the chunk from the given string. To load and run a given string, use the idiom

```
assume(loadstring(s))()
```

**map (f, obj [, ...])**

This operator maps a function `f` to all the values in table, set, sequence, 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 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: **remove**, **select**, **subs**, **zip**.

**mapto**set (f, obj [, ...])

Maps a function *f* to all the values in table or sequence *obj* and returns a set. Metamethods, if existing, are not copied. See **map** for further information.

**max** (obj [, 'sorted'])

Returns the maximum of all numeric values in table or sequence *obj*. If the option 'sorted' is passed then 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**.

**min** (obj [, 'sorted'])

Returns the minimum of all numeric values in table or sequence *obj*. If the option 'sorted' is passed then 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**.

**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.

**nseq (a , b [, step])**

Creates a sequence **seq**(a, a+step, ..., b-step, b), with a, b, and step being numbers. The step size is 1 if step is not given.

The function uses the Kahan summation algorithm to prevent round-off errors. Thus, the return is much more precise with step sizes that are not integers than iterations with for/to loops are.

See also: **calc.fseq**.

**ops (index, ...)**

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, ?)`).

**pcall (f, arg1, ...)**

Calls function *f* with the given arguments in *protected mode*. This means that any error inside *f* is not propagated; instead, pcall catches the error and returns a status code. Its first result is the status code (a boolean), which is true if the call succeeds without errors. In such case, pcall also returns all results from the call, after this first result. In case of any error, pcall returns **false** plus the error message.

**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.



**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 Agenda, **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 Agenda does not print a final newline when finishing output. Note that these two options cannot be used together.

If the kernel setting `kernel('longtable')` is set to **true**, then each key~value pair is printed on a separate line, and Agenda halts after `_Env.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 `_Env.More` is 40 lines, but you may change this value in the Agenda session or in the Agenda initialisation file.

You may change the way **print** formats objects by changing the respective `_EnvPrint` functions in the `library.agn` file. See Appendix A5 for further details.

**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.

**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 length of the table or sequence, so that a call `put(obj, x)` inserts `x` 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.

**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.

**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`.

**rdelete (f)**

Deletes the remember table or read-only remember table of procedure `f` entirely. The function returns **null**.

**read (fn)**

Reads an object stored in the binary file denoted by file name `fn` 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/adena/lib`.

In OS/2 and Windows, the Agena installation program 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:\adena\lib` folder and optionally in the `d:\adena\mypackage` folder.

```
SET AGENAPATH=d:/adena/lib  OR
SET AGENAPATH=d:/adena/lib;d:/adena/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/adena/lib` folder and optionally in the `/home/usr/adena/mypackage` folder.

```
SET AGENAPATH=/home/usr/adena/lib  OR
SET AGENAPATH=/home/usr/adena/lib;/home/usr/adena/mypackage
```

In DOS, you have to set **AGENAPATH** in the `autoexec.bat` file:

```
SET AGENAPATH=d:/adena/lib  OR
SET AGENAPATH=d:/adena/lib;d:/adena/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**.

**register (packagename, name1 [, name2, ...])**

Defines short names for a package. It enters the strings `name1` (and `name2`, etc., if given) into the table **pkgname.loaded**, so that if you initialise a package with the **with** function, those names `namex` can be used as short names for package functions instead of the fully written function names.

Thus instead of later calling a function by ``packagename.name(arguments)`` you may use the shortcut ``name(arguments)``. See **with** for more details.

This is short for insert `name1 [, name2, ...]` into `packagename.loaded`. If a name is already included in the table, **register** does not add it.

### **`_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 `_Env.Release`.

```
remember (f)
remember (f, tab)
remember (f, null)
```

Administers remember tables.

In the first form, the remember table stored to procedure `f` is returned. See **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:

```
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 table.

**remember** always returns **null**. It is written in the Agena language and included in the `library.agn` file.

See chapter 6.14 for examples. See also: **defaults**.

**remove (f, obj [, ...])**

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.

See also: **map**, **select**, **subs**, **zip**.

**restart**

Restarts an Agenda session. No argument is needed.

During start-up, Agenda 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 Agenda session is restarted with **restart**, all values in the Agenda environment are unassigned including the environment variable **\_G**, but except of **\_origG**, **homedir**, **mainlibname**, and **libname** (**mainlibname** and **libname** are reset to their original values if the kernel setting `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 `agenda.lib` and thereafter the initialisation file `agenda.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**.

**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;

> 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**.

#### **rright (obj)**

With the pair *obj*, the operator returns its right operand.

See also: **left**.

#### **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**.

#### **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**.

#### **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]).
```

#### **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**.

**save (obj, fn)**

Saves an object `obj` of any type into a binary file denoted by file name `fn`.

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

**save** returns an error if an object that cannot be stored to a file has been passed: procedures, threads, userdata, for example. It also returns an error if the object to be written is self-referencing (e.g. `_G`).

Note that **save** overwrites existing files without warning.

See also: **read**.

**select (f, obj [, ...])**

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.

See also: `countitems`, **map**, **remove**, **subs**, **zip**.

**setbit (x, pos, bit)**

Sets or unsets a bit in a negative or positive integer `x`.

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.

**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.

**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`.

**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.

Note that the `try` statement does not handle user-defined types.

See also: `gettype`.

**size (obj)**

With tables, the operator returns the number of key~value pairs in table `obj`.

With sets 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.

**sort (obj [, comp])**

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 `comp` 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 `comp(a[i+1], a[i])` will be **true** after the sort). If `comp` 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.

Example:

```
> s := [1, 2, 3]
> sort(s, << x, y -> x > y >>)
> s:
[3, 2, 1]
```

**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: **map**, **remove**, **select**, **zip**.

**time ()**

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

**top (obj)**

With the sequence `obj`, the operator returns the element with the largest index. If the sequence 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: **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.

**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.

**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**.

**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 global **infolevel** table along with a *level* (an integer) indicating the **infolevel** setting at which information will be printed.

If you do not enter an entry for the function to the **infolevel** table, then nothing is printed.

```
> f := proc(x) is
>   userinfo(f, 1, 'this is a primary info to the user: ', x);
>   userinfo(f, 2, 'this is an additional info to the user: ', x)
> end;
```

If the *level* argument to **userinfo** is equal or less than the **infolevel** table setting, then the information is printed, otherwise nothing is printed.

```
> infolevel[f] := 2;

> f('hello !');
this is a primary info to the user:      hello !
this is an additional info to the user:  hello !
```

Now the **infolevel** is decreased such that less information will be output.

```
> infolevel[f] := 1;

> f('hello !');
this is a primary info to the user:      hello !
```

**whereis** (obj, x)

Returns the indices for a given value *x* in table or sequence *obj* as a new table or sequence, respectively. The function is written in the Agena language and included in the **library.agn** file.

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.

You may optionally assign short names to either all or only specific procedures. If you only want define short names to some of the functions, define a table `packagename.loaded` and include the respective function names as strings. If the table `packagename.loaded` is not present, **with** assigns short names to all keys in `packagename`.

Note that if `packagename.name` is not of type procedure, a short name is not created for this object.

If there is a table `packagename.loaded`, then **with** prints only those values included in this table. If `packagename.loaded` does not exist, all keys in `packagename` are printed.

An example: If your package is called ``agenapackage``, then the short names to be printed are included in:

```
agenapackage.loaded := ['run', 'dosomething'];
```

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 \
December 24, 2008\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 `_EnvWithVerbose` 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 `_EnvProtected`), 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 `_EnvProtected`.

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 a sequence of 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**.

```
writeline ([fh,] v1 [, v2 ...] [, delim = <str>])
```

This function prints a sequence of 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`.

```
xpcall (f, err)
```

This function is similar to `pcall`, 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`.

```
zip (f, s1, s2)
```

This function zips together either two sequences or two tables `s1`, `s2` by applying the function  $f$  to each of its respective elements. The result is a new sequence or table  $s$  where each element  $s[k]$  is determined by  $s[k] := f(s1[k], s2[k])$ .

`s1` and `s2` must have the same number of elements. If you pass tables, they must have the same keys.

If `s1` or `s2` have user-defined types or metatables, they are copied to the resulting structure, as well. If `s1` has a metatable, then this metatable is copied, else the metatable of `s2` is used if the latter exists. The same applies to user-defined types.

See also: `map`, `remove`, `select`, `subs`.

## 7.2 Coroutine Manipulation

The operations related to coroutines comprise a sub-library of the basic library and come inside the table `coroutine`.

### `coroutine.create (f)`

Creates a new coroutine, with body `f`. `f` must be a Agena function. Returns this new coroutine, an object with type 'thread'.

### `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.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 a 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.



## 7.3 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.loaded**

A table with all 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.

## 7.4 String Manipulation

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	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	đ	Đ	þ	Þ	ß					

### 7.4.1 Kernel Operators and Basic Library Functions

**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.

**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. Like **atendof**, the operator also returns **null** if the strings have the same length or at least one of them is the empty string.

**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.

**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**])

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**. 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, almost similar to regular expressions, see chapter 7.4.3. **instr** is 45 % faster than **strings.find**.

See also: `in` operator, `strings.find`, `strings.seek`.

#### `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.

#### `replace (s1, s2, s3)`

#### `replace (s1, struct)`

#### `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 `struct` 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.

See also: `strings.gsub`.

#### `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 **kernel** for further information.

#### **trim (s)**

Returns a new string with all leading, trailing and excess embedded white spaces removed. **trim** is an operator.

#### **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.

### 7.4.2 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.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'):
AEIOUÄÖÜÆØ
```

Note that the function does not convert all existing special tokens.

Agenda is shipped with substitution tables for codepage 1252. If you want to use another codepage, edit the `_c2f` and `_f2c` tables in the `lib/library.agn` file accordingly.

**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 7.4.3), 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` and `instr` operator, `strings.seek`, `strings.rseek`.

**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 Agenda 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 \n 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.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 7.4.3.

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` 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)**

Returns the number of occurrences of substring `pattern` in string `s`. The function does not support pattern matching expressions.

#### **strings.isAbbrev (s, pattern [, true])**

Determines whether a string `s` is abbreviated by 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 the length of `s`. The function returns **true** or **false**.

If only two arguments are passed, pattern matching facilities (see chapter 7.4.3) 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**.

See also: **atendof**, **strings.isEnding**.

**strings.isAlpha (s)**

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

See also: **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**.

**string.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.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 7.4.3) 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.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 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.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.isNumberSpace (s)**

Checks whether the string *s* consists entirely of the digits 0 to 9 or white spaces and returns **true** or **false**.

**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.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.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.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 does not support pattern matching facilities.

See also: **strings.gmatch**.

**strings.put (s1, n, s2)**

The function has been deprecated. Use the **replace** operator instead.

**strings.remove (s, pos, len)**

Starting from string position *pos*, the function removes *len* characters from string *s*. The return is a new string.

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.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.rseek (s, pattern [, init])**

Starting from the right end and always running to its left beginning, the function looks for the first match of *pattern* in the string *s*. If it finds a match, then **rseek** returns the index of *s* where this occurrence starts with respect to its left beginning; otherwise, it returns **null**.

A third, optional numerical argument *init* specifies where to start the search; its default value is **size pattern** and may be negative. If *init* is positive, the search begins from the *init*'s character from the left (and also runs to the left). If *init* is negative, the search begins from the *|init|*'s character from the right (and runs to the left, also).

The function is useful for example in linguistic research to search for inflectional endings. The function does not support pattern matching facilities.

See also: **in** and **instr** operators, **string.find**, **strings.seek**.

**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.ltrim**.

**strings.seek (s, pattern [, init])**

Looks for the first match of *pattern* in the string *s*. If it finds a match, then **seek** returns the index of *s* where this occurrence starts; 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. Contrary to **strings.find**, the function does not support pattern matching facilities but is around 8 % faster. If you have to search a string from its beginning, use the faster **in** operator.

See also: **in** and **instr** operators, **strings.find**, **strings.rseek**.

**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.words (s [, delim])**

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. The return is a number.

### 7.4.3 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.
- **%p**: represents all punctuation characters.
- **%s**: represents all space characters.
- **%u**: represents all uppercase letters.
- **%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, 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);
- %bxy, 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.

## 7.5 Table Manipulation

### 7.5.1 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. If the table contains tables itself, those tables are also copied properly (by a `deep copying` method). Metatables and user-defined types are copied, too.

#### **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 function works with dictionaries, as well.

#### **getentry** (o [, k<sub>1</sub>, ..., k<sub>n</sub>])

Returns the entry *o*[*k*<sub>1</sub>, ..., *k*<sub>*n*</sub>] from the table *o* without issuing an error if one of the given indices (second to last argument) does not exist.

#### **join** (t)

Concatenates all string values in the table *t* in sequential order and returns a string.

#### **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: **select**, **remove**, and **subs**.

#### **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.

**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.

**unique (t)**

The **unique** operator removes all holes (‘missing keys’) in a table `t` and removes multiple occurrences of the same value, if present. The return is a new table with the original table unchanged.

**zip (f, t1, t2)**

This function zips together either two tables `t1`, `t2` by applying the function `f` to each of its respective elements. See Chapter 7.1 for more information.

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.  $\{1, 1\} = \{1\} \rightarrow \text{true}$ ,  $\{1, 2\} \text{ xsubset } \{1, 1, 2, 2, 3, 3\} \rightarrow \text{true}$ .

**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 `table2` 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 intersect t2**

Searches all values in `t1` that are also values in `t2` and returns them in a new table. The search is quadratic, so you may use **bintersect** instead if you want to compare large tables since **bintersect** performs a binary search.

**t1 minus t2**

Searches all values in table `t1` that are not values in table `t2` and returns them as a new table. The search is quadratic, so you may use **bminus** instead if you want to compare large tables since **bminus** performs a binary search.

**t1 subset t2**

Checks whether all values in table `t1` are included in table `t2` and returns **true** or **false**. The operator also returns **true** if `t1 = t2`. The search is quadratic.

**t1 union t2**

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

**t1 xsubset t2**

Checks whether all values in table `t1` are included in table `t2` and whether `t2` contains at least one further element, so that the result is always **false** if `t1 = t2`. The search is quadratic.

See also: **bintersect**, **bisEqual**, **bminus**, **countitems**, **purge**, **put**, **remove**, **select**, **sort** in Chapter 7.1 Basic Functions.

## 7.5.2 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*<sub>1</sub>, *value*<sub>1</sub> [, *key*<sub>2</sub>, *value*<sub>2</sub>, ..., *key*<sub>*n*</sub>, *value*<sub>*n*</sub>])

Sets the specified keys and values to table *t*, i.e. *t*[*key*<sub>*k*</sub>] := *value*<sub>*k*</sub>. 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**.

**tables.indices** (*t*)

Returns all keys of table *t* in a 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.)

## 7.6 Set Manipulation

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. If the set contains other sets - even nested ones-, those sets are also copied properly (by a `deep copying` method). Metamethods if present, are also copied.

### `filled (s)`

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

### `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.  $\{1, 1\} = \{1\} \rightarrow \text{true}$ ,  $\{1, 2\} \times \text{subset } \{1, 1, 2, 2, 3, 3\} \rightarrow \text{true}$ .

### `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 minus s2**

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

**s1 subset s2**

Checks whether all items in set `s1` are included in set `s2` and returns **true** or **false**. The operator also returns **true** if `s1 = s2`. The search is linear.

**s1 union s2**

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

**s1 xsubset s2**

Checks whether all items in set `s1` are included in set `s2` and whether `s2` contains at least one further item, so that the result is always **false** if `s1 = s2`. The search is linear.

## 7.7 Sequence Manipulation

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. If the sequence contains sequence itself, those sequence are also copied properly (by a `deep copying` method). Metatables and user-defined types are copied, too.

### `filled (s)`

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

### `getentry (s [, k1, ..., kn])`

Returns the entry `s[k1, ..., kn]` from the sequence `s` without issuing an error if one of the given indices (second to last argument) does not exist.

### `join (s)`

Concatenates all string values in the sequence `s` in sequential order and returns a string.

### `map (f, s [, ...])`

Maps the function `f` on all elements of a sequence `s`. See `map` in chapter 7.1 for more information.

### `qsadd (s)`

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

### `sadd (s)`

Sums up all numeric values in sequence `s`. The return is a number. If `s` is empty or consists entirely of non-numbers, **null** is returned. If `s` contains numbers and other values, only the numbers are added.

### `size (s)`

Returns the number of items in a sequence `s`.

**unique (s)**

With a sequence *s*, the **unique** operator removes multiple occurrences of the same item, if present in *s*. The return is a new sequence with the original sequence unchanged.

**zip (f, s1, s2)**

This function zips together either two sequences *s1*, *s2* by applying the function *f* to each of its respective elements. See Chapter 7.1 for more information.

See also: **bintersect**, **bisEqual**, **bminus**, **countItems**, **purge**, **put**, **remove**, **select**, **sort** 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.  $\text{seq}(1, 1) = \text{seq}(1) \rightarrow \text{true}$ ,  $\text{seq}(1, 2) \text{ xsubset } \text{seq}(1, 1, 2, 2, 3, 3) \rightarrow \text{true}$ .

**s1 == s2**

This equality check of two sequences *s1*, *s2* first tests whether *s1* and *s2* point to the same sequence reference in memory. If so, it returns **true** and quits.

If not, the operator then checks whether *s1* and *s2* contain the same values without regard to their keys, and returns **true** or **false**. In this case, the search is quadratic.

**s1 == s2**

This strict equality check of two sequences *s1*, *s2* first tests whether *s1* and *s2* point to the same sequence reference in memory. If so, it returns **true** and quits.

If not, the operator then checks whether *s1* and *s2* contain the same number of elements and whether all entries in the sequences are the same and are in the same order, and returns **true** or **false**. In this case, the search is linear.

**s1 <> s2**

This inequality check of two sequences *s1*, *s2* first tests whether *s1* and *s2* do not point to the same sequence reference in memory. If so, it returns **true** and quits.

If not, the operator then checks whether *s1* and *s2* do not contain the same values, and returns **true** or **false**. In this case, the search is quadratic.

**c in s**

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

### **s1 intersect s2**

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

### **s1 minus s2**

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

### **s1 subset s2**

Checks whether all values in sequence `s1` are included in sequence `s2` and returns **true** or **false**. The operator also returns **true** if `s1 = s2`. The search is quadratic.

### **s1 union s2**

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

### **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 <b>intersect</b> operator but much faster with very large sequences
bisEqual	same as the <code>=</code> operator but much faster with very large sequences
bminus	same as the <b>minus</b> operator but much faster with very large sequences
duplicates	returns all the values that are stored more than once in the given sequence

## 7.8 Mathematical Functions

### 7.8.1 Kernel Operators

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

**$x \ \&\& \ y$**

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: **kernel** in Chapter 7.1.

**$\sim\sim \ (x)$**

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 **kernel** function with the **signedbits** option. See also: **kernel** in Chapter 7.1.

**$x \ \_ \_ \ y$**

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 **kernel** function with the **signedbits** option. See also: **kernel** in Chapter 7.1.

**$x \ \_ \^ \ y$**

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 **kernel** function with the **signedbits** option. See also: **kernel** in Chapter 7.1.

**abs  $(x)$**

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

**arccos  $(x)$**

The inverse cosine function ( $x$  in radians). Complex numbers are supported.

**arcsin  $(x)$**

The inverse sine function ( $x$  in radians). Complex numbers are supported.

**arctan  $(x)$**

The inverse tangent function ( $x$  in radians). Complex numbers are supported.

**cos  $(x)$**

Returns the cosine of  $x$  ( $x$  in radians). Complex numbers are supported.



**cosh (x)**

Returns the hyperbolic cosine of  $x$ . Complex numbers are supported.

**entier (x)**

Rounds  $x$  downwards to the nearest integer. Complex numbers are supported.

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

**even (x)**

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

**exp (x)**

Exponential function, returns the value  $e^x$ . Complex numbers are supported.

**finite (x)**

Checks whether  $x$  is a number and neither  $\pm\text{infinity}$  nor **undefined** (NaN). It 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**.

**lngamma (x)**

Computes  $\ln \Gamma x$ . If  $x$  is a non-positive number, the function returns **undefined**. Complex numbers are supported.

See also: **gamma** function.

**gethigh (x)**

Returns the higher bytes of a number  $x$  as an integer. See also: **getlow**.

**getlow (x)**

Returns the lower bytes of a number  $x$  as an integer. See also: **gethigh**.

**int (x)**

Rounds  $x$  to the nearest integer towards zero. Complex numbers are supported.

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

**ln (x)**

Natural logarithm of  $x$ . If  $x$  is non-positive, the function returns **undefined**. Complex numbers are supported.

**sethigh (x, i)**

Sets the higher bytes of the number  $x$  to the integer  $i$ , and returns the new number. See also: **setlow**.

**setlow (x, i)**

Sets the lower bytes of the number  $x$  to the integer  $i$ , and returns the new number. See also: **sethigh**.

**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 **kernel** function with the **signedbits** option. See also: **kernel**.

**sign (x)**

Determines the sign of the number or complex value  $x$ . If  $x$  is a complex value, the result is determined as follows:

- 1, if  $\text{real}(x) > 0$  or  $\text{real}(x) = 0$  and  $\text{imag}(x) > 0$
- -1, if  $\text{real}(x) < 0$  or  $\text{real}(x) = 0$  and  $\text{imag}(x) < 0$
- 0 otherwise.

**sin (x)**

Returns the sine of  $x$  ( $x$  in radians). Complex numbers are supported.

**sinh (x)**

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 function returns **undefined**.

With complex numbers, the operator returns the complex square root, in the range of the right halfplane including the imaginary axis.

**tan (x)**

Returns the tangent of  $x$  ( $x$  in radians). Complex numbers are supported.

**tanh (x)**

Returns the hyperbolic tangent of  $x$  ( $x$  in radians). Complex numbers are supported.

## 7.8.2 Base Library Functions

The following mathematical functions are provided as part of the base library.

### **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 ( $|x-y| \leq \text{eps}$ ) suited for values `near` 0 and a simplified relative approximation algorithm developed by Donald H. Knuth suited for larger values ( $|x-y| \leq \text{eps} * \max(|x|, |y|)$ ), 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.

### **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.

### **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.

**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.

**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.

**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.

**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  $x \geq 0$ , and  $\pi$  otherwise.

**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  $x-ly$  of the complex value  $z=x+ly$ . If  $z$  is of type number, it is simply returned.

### **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.

### **erf (x)**

Returns the error function of  $x$ . It is defined by  $\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$ . The function works on both numbers and complex values.

### **erfc (x)**

Returns the complementary error function of  $x$ , a number or complex value. It is defined by  $\text{erfc}(x) = 1 - \text{erf}(x)$ . The return is a number or complex value.

**exp2 (x, sign)**

Computes either  $e^{x^2}$  if  $\text{sign} \geq 0$ , or  $e^{-x^2}$  if  $\text{sign} < 0$  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.

**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 which you may extend by editing the `library.agn` file.

**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.  $x - \text{int}(x)$ . The function is implemented in the Agena language and included in the `library.agn` file.

**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).

**gamma (x)**

The gamma function  $\Gamma x$ .  $x$  may be a number or complex value.

See also: **lngamma** operator.

**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.

### **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  $x*r \geq 0$ .

### **ldexp (m, e)**

Returns  $m2^e$  ( $e$  should be an integer).

### **log (x, b)**

Returns the logarithm of the number or complex number  $x$  to the base  $b$ , with  $b$  a number or a complex number. The function is implemented in the Agena language and included in the `library.agn` file.

### **log10 (x)**

Returns the base-10 logarithm of the number or complex number  $x$ . The function is implemented in the Agena language and included in the `library.agn` file.

### **modf (x)**

Returns two numbers, the integral part of  $x$  and the fractional part of  $x$ .

### **root (x, n)**

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

### **roundf (x [, d])**

Rounds the number  $x$  to the  $d$ -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.

**7.8.3 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  $\epsilon := |(x - n/d) / x| \leq \text{err}$ . 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.

**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.

**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.

**math.max (x, ...)**

Returns the maximum value among its arguments.

**math.min (x, ...)**

Returns the minimum value among its arguments.

**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.

**math.prevPrime (x)**

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

See also: **math.nextPrime**, **math.isPrime**.

**math.Phi**

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

**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  $2 * \text{\_Env.MaxLong}$  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. The return are the 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.

**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 Input and Output Facilities

The I/O library provides two ways for file manipulation:

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:/agenda/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:/agenda/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, and Windows editions only. On other systems, it returns **fail**.

See also: `io.getkey`.

```
io.close ([filehandle])
```

Closes a file. 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.

See also: **io.open**, **io.popen**.

```
io.flush (filehandle)
```

```
io.flush ()
```

In the first form, saves any written data to `filehandle`. In the second form, the function flushes the default output.

```
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.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.

```
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 **nil** (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 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.

```
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. The function does not lock the file (see **`io.lock`**).

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` 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.pcall`.

**`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.
- **'\*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 and returns a string or number.

```
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.

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.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.

**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.flush**)).
- **'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, `size` specifies the size of the buffer, in bytes. The default is an appropriate size.

**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)
```

```
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 (...)
```

Write the value of each of its arguments to standard output if the first argument is not a file handle, or to the file denoted by the first argument (a file handle). Except for the file handle and the 'delim' option described below, all arguments must be strings or numbers. To write other values, use **toString** or **strings.format** before using **write** or **writeline**.

**writeline** adds a new line character 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>`.

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',
> 'bablen', 'anne', 'Küst', 'nix', 'anneres', 'as', 'Platt.')
Betto'n16.JohrhunnertgeefdathierbablenanneKüstnixanneresasPlatt.
```

Use a white space as a separator:

```
> io.writeline('Bet', 'to\n', '16.', 'Johrhunnert', 'geef', 'dat', 'hier',
> 'bablen', 'anne', 'Küst', 'nix', 'anneres', 'as', 'Platt.',
> delim~' ')
Bet to'n 16. Johrhunnert geef dat hier bablen 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', 'babben', 'anne', 'Küst', 'nix', 'anneres', 'as', 'Platt.',
>   delim~' '):
true
```

Finally, the file will be closed.

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

## 7.10 binio - Binary File Package

This package contains functions to read data from and write data to binary files.

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:/agenda/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:

**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.) 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 **fail**.

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 `_Env.BufferSize` bytes from the file denoted by `filehandle` and returns them as a sequence of integers. You may change the value of `_Env.BufferSize` to any other positive integer in order to read less or more 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 the 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 Agenda 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.write\_number** function.

See also: **binio.write\_number**.

**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 file handle. The function returns **true** if successful, 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  $0 \leq \text{number} < 256$ , 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` to the file denoted by `filehandle` at its current position. The function returns **true** in case of success and **quits with an error** otherwise.

The `number` should be an integer with  $0 \leq \text{number} < 256$ , otherwise `number % 256` will be stored to the file.

Internally, the byte is stored as a C `unsigned char`.

**binio.writelong (filehandle, number)**

The function writes the given Agena `number` to the file denoted by `filehandle` at its current position. The `number` should be an integer with `_Env.MinLong < number < _Env.MaxLong`, otherwise the operations is not defined.

The function returns **true** in case of success and quits with an error otherwise.

Internally, the number is stored as a 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` to the file denoted by `filehandle` at its current position. The function returns **true** in case of success and issuesd an error otherwise. The `number` is always stored in Big Endian notation. The **binio.readnumber** function makes proper conversion to Little Endian if Agena runs on a Little Endian machine.

**binio.writeshortstring** (*filehandle*, *string*)

The function writes the given *string* to the file denoted by *filehandle* at its current position. The string can be of length 0 to 255.

The function returns **true** in case of success and issues an error otherwise. Internally, **writeshortstring** first writes the length of the string as a C unsigned char 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.readshortstring**.

**binio.writestring** (*filehandle*, *string*)

The function writes the given *string* 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 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**.



## 7.11 Operating System Facilities

This library is implemented through table `os`.

### `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'	<b>true</b> if a battery is present, and <b>false</b> 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'	<b>true</b> if battery is currently being charged, or <b>false</b> otherwise
'flag'	the battery flag, a number
'lifetime'	the remaining battery lifetime in seconds, a number (or <b>undefined</b> if it could not be determined)
'fulllifetime'	the battery lifetime in seconds when at full charge, a number (or <b>undefined</b> 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'	<b>true</b> if power management is switched on, or <b>false</b> if not.

On other operating systems, the function returns **fail**.

### `os.alldirs (str)`

Returns the names of all directories on the file system that are part of the folder `str`, with `str` a string. **The return is a table with all the path names.**

### `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 is activated `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.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.chdir (str)**

Changes into the directory given by string `str` on the file system. Returns **true** on success, and **fail**, the error message from the operating system, and the C error code otherwise.

#### **os.curdir ()**

Returns the current working directory on the file system as a string or **fail** if the path could not be determined.

#### **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), `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, `date` returns a reasonable date and time representation that depends on the host system and on the current locale (that is, `os.date()` is equivalent to `os.date('%c')`).

#### **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`.

### **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, ...)
'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.

### **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.popen**.

### **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)**

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' - readonly flag

The function returns **true** on success, and **fail** otherwise.

Examples:

```
> chmod('file.txt', 'a-wx'); # deletes write and execute permissions
```

See also: **os.fstat**.

**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 `_Env.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** otherwise. It also returns **fail** if the file could be copied, but the file permissions could not be set.

#### **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: **used**.

#### **os.fstat (fn)**

Returns information on the file, link (UNIX only), or directory given by the string *fn* in a table.

The table includes the following information:

Key	Meaning
'mode'	'FILE' if <i>fn</i> is a regular file, 'LINK' if <i>fn</i> is a symbolic link (UNIX only), 'DIR' if <i>fn</i> is a directory, 'CHARSPECFILE' if <i>fn</i> is a character special file (a device like a terminal), 'BLOCKSPECFILE' if <i>fn</i> 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 mode <i>t</i> )
'bits'	<p>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.</p> <p>The letters indicate:</p> <ul style="list-style-type: none"> <li>'r' - read permission granted (UNIX &amp; DOS)</li> <li>'w' - write permission granted (UNIX &amp; DOS)</li> <li>'x' - execute permission granted (UNIX &amp; DOS)</li> <li>'d' - indicates directory (OS/2 only)</li> <li>'r' - readonly file (OS/2 and Windows)</li> <li>'h' - hidden file (OS/2 and Windows)</li> <li>'a' - archived file (OS/2 and Windows)</li> <li>'s' - system file (OS/2 and Windows)</li> </ul>

'owner', 'group', 'other'	<p>Access permissions to the file or directory are returned with the <code>owner</code>, <code>group</code> (UNIX only), and <code>other</code> (UNIX only) keys which each reference tables with information on <code>read</code>, <code>write</code>, and <code>execute</code> permissions. These tables have the following form: <code>['read' ~ &lt;boolean&gt;, 'write' ~ &lt;boolean&gt;, 'execute' ~ &lt;boolean&gt;]</code>, where <code>&lt;boolean&gt;</code> is either <b>true</b> or <b>false</b>.</p> <p>In OS/2 and Windows, the file attributes <code>'hidden'</code>, <code>'readonly'</code>, <code>'archived'</code>, and <code>'system'</code> are also returned in the subtable with key <code>'owner'</code>.</p>
---------------------------------	--

See also: `os.fattrib`.

#### `os.getenv (varname)`

Returns the value of the process environment variable `varname`, or **null** if the variable is not defined.

#### `os.isANSI ()`

Returns **true** on Agena editions compiled with the `LUA_ANSI` (strict ANSI C) option, and **false** otherwise.

#### `os.isDOS ()`

Returns **true** if Agena is run in DOS, and **false** otherwise. It also returns **false** if Agena is run from within a Windows shell.

#### `os.isHaiku ()`

Returns **true** if Agena is run in Haiku, and **false** otherwise.

#### `os.isLinux ()`

Returns **true** if Agena is run in Linux, and **false** otherwise.

#### `os.isMac ()`

With no options, returns **true** if Agena is run on a Mac, and **false** otherwise.

With the option `'ppc'`, the function determines whether Agena is run on a PowerPC CPU and returns **true** or **false**.

With the option `'x86'` or `'intel'`, the function determines whether Agena is run on an Intel CPU and returns **true** or **false**.

**os.isOS2 ()**

Returns **true** if Agenda is run in OS/2, and **false** otherwise.

**os.isSolaris ([option])**

With no options, returns **true** if Agenda is run in Solaris (including Nexenta), and **false** otherwise.

With the option 'sparc' or 'sun4u', the function determines whether Agenda is run on Sun Sparc Solaris and returns **true** or **false**.

With the option 'x86' or 'intel', the function determines whether Agenda is run on Sun x86 Solaris and returns **true** or **false**.

**os.isUNIX ()**

Returns **true** if Agenda is run in a UNIX environment (i.e. Solaris, Linux, and Nexenta), and **false** otherwise.

**os.isWin ()**

Returns **true** if Agenda is run in Windows, and **false** otherwise.

**os.list (d [, options])**

Lists the contents of a directory *d* (given as a string) as a table. If *d* is void, the current working directory is evaluated.

*d* may include the ? and \* jokers known from UNIX, OS/2, or Windows to select a subset of files, e.g. `os.list('*.c')` to select 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, files are returned. If the optional argument 'dirs' is given, directories are returned. If the optional argument 'links' is given, links are returned (UNIX only). Multiple options can be given.

If *d* is '.', then the current working directory is examined. If *d* is '..', then the directory one level higher is searched.

**os.listcore (d)**

**os.listcore (d [, options])**

In the first form, returns a table with all the files, links and directories in the given path *d*. If *d* is void, the current working directory is evaluated.

In the second form, by giving at least one of the options 'files', 'dirs', or 'links', file, directory name, or link names are returned, respectively. These three

options can be mixed.

#### **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
'wiredown'	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**.

#### **os.mkdir (str)**

Creates a directory given by string `str` on the file system. Returns **true** on success, and fail, the error message from the operating system, and the C error code 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. If this function fails, it returns **fail**, the error message from the operating system, and the C error code otherwise.

#### **os.remove (filename)**

Deletes the file or directory with the given name. Directories must be empty to be removed. Returns **true** on success, and **fail**, the error message from the operating system, and the C error code otherwise.

#### **os.rmdir (dirname)**

Deletes a directory denoted by the string `dirname` on the file system. Returns **true** on success, and **fail**, the error message from the operating system, and the C error code 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.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, this function only returns the name of the current locale for the given category.

#### **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 Number (`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**.

**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**.

**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**.

## 7.12 The Debug Library

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`.

### **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).

**debug.getmetatable** (*object*)

Returns the metatable of the given *object* or `null` if it does not have a metatable.

**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.

**debug.setfenv** (*object*, *t*)

Sets the environment of the given *object* to the given table *t*. Returns *object*.

**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.

**debug.setmetatable** (object, t)

Sets the metatable for the given `object` to the given table `t` (which can be **null**).

**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.

**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.

## 7.13 utils - Utilities

The **utils** package provides miscellaneous functions.

**utils.arraysize (strarr)**

Returns the maximum number of elements allocable to the `stringarray` userdata denoted by **strarr**.

See also: **utils.newarray**.

**utils.getarray (strarr, n)**

Returns the  $(n+1)$ -th string from the `stringarray` userdata denoted by **strarr**. Note that **n** starts from 0.

See also: **utils.newarray**.

**utils.getwholearray (strarr)**

Returns a table including all strings that are stored in the `stringarray` userdata denoted by **strarr**, with the first string at table index 1 (and not 0).

See also: **utils.newarray**.

**utils.newarray (n)**

Creates a `stringarray` userdata of exactly **n** strings,  $n > 0$ . The userdata stores (C pointers to) strings of any size, including empty strings. The strings can be set into the userdata by the **utils.setarray** function and accessed through the **utils.getarray** function.

**utils.setarray (strarr, n, str)**

Sets the string **str** into the `stringarray` userdata denoted by **strarr** at position **n**. Note that **n** starts from 0, so your first string must be stored to index 0 of the userdata.

See also: **utils.newarray**.

**utils.singlesubs (str, strarr)**

Substitutes individual characters in string **str** by corresponding replacements in the `stringarray` userdata denoted by **strarr**. 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 the userdata. If an entry is found for index **n**, then the character is replaced, otherwise the character remains unchanged.

See also: **utils.newarray**.

Other functions in the **utils** library are:

**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**.

**utils.isLeapYear (x)**

Returns **true** if the given year *x* (a number) is a leap year, and **false** otherwise.

**utils.readcsv (filename [, options])**

Reads a 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. If a line contains only one field, then it is returned without including it in a sequence. If a line contains nothing, i.e. '\n', then an empty string is returned.

Strings containing numbers are converted to numbers.

Options can be passed as pairs:

Left pair element	Right pair element	Example
delim	A string. Use this string as the delimiter instead of a semicolon.	delim ~ ' '
skipemptylines	<b>true</b> or <b>false</b> : If <b>true</b> , do not return empty lines. Default is <b>true</b> .	skipemptylines ~ true
skipspaces	<b>true</b> or <b>false</b> : If <b>true</b> , do not return lines including spaces only. Default is <b>false</b> .	skipspaces ~ true

Left pair element	Right pair element	Example
<code>ignorespaces</code>	all spaces in a line are deleted before returning the fields. Default is <b>true</b> .	<code>ignorespaces ~ true</code>

The function is written in the Agena language and included in the `lib/utils.agn` file.

See also: `utils.writecsv`.

**`utils.writecsv (o, filename [, delim [, keyoption]])`**

Creates a CSV file. The function writes all values or keys and value(s) of a table, set, or sequence `o` 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 is a structure itself, then all entries in this substructure are separated by the given delimiter; default is a semicolon.

If the optional argument `keyoption` is given, 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.

The function is written in the Agena language and included in the `lib/utils.agn` file.

See also: `utils.readcsv`.



## 7.14 stats - Statistics

This package contains procedures for statistical calculations and operates completely on tables. As a *plus* package, it is not part of the standard distribution and must be activated with the **readlib** or **with** functions.

**stats.median (t)**

Returns the median of all numeric values in table *t* as a number.

**stats.mean (t)**

Returns the mean of all numeric values in table *t* as a number. The function is implemented in Agena and included in the `library.agn` file.

**stats.minmax (t [, 'sorted'])**

Returns a table with the minimum of all numeric values in table *t* as the first value, and the maximum as the second value. If the option 'sorted' is passed then the function assumes that all values in *t* are sorted in ascending order so that execution is much faster.

**minmax** returns **fail** if a sequence or table of less than two elements has been passed.

**stats.qmean (t)**

Returns the quadratic mean of all numeric values in table *t* as a number. The function is implemented in Agena and included in the `library.agn` file.

**stats.sd (t)**

Returns the standard deviation of all numeric values in table *t* as a number. The function is implemented in Agena and included in the `library.agn` file.

**stats.toVals (t)**

Converts all string values in table *t* to Agena numbers. The function is implemented in Agena and included in the `library.agn` file.

**stats.var (t)**

Returns the variance of all numeric values in *t* as a number. The function is implemented in Agena and included in the `library.agn` file.

## 7.15 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.

A typical example might look like this:

```
> readlib 'calc'
```

Define a function  $f := x \rightarrow \sin(x)$ :

```
> 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):
0.999999999999963      1.8503717573394e-010
```

Compare it:

```
> cos(0):
1
```

Integrate it over  $[0, \pi]$ :

```
> 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.dawson (x)**

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

See also: **exp2**.

**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  $f(a), \dots, f(b)$ , with  $f$  a function,  $a$  and  $b$  numbers. If  $a > b$ , then the result is 1.

**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])**

Creates a sequence **seq**( $1 \sim f(a), 2 \sim f(a+step), \dots, ((b-a)*1/step+1) \sim f(b)$ ), with  $f$  a function,  $a$  and  $b$  numbers. Thus, the function  $f$  is applied to all numbers between and including  $a$  and  $b$ . The step size is 1 if  $step$  - a number - is not given.

The function uses the Kahan summation algorithm to prevent round-off errors.

See also: `nseq`, `calc.fsum`.

**calc.fsum (f, a, b)**

Computes the sum of  $f(a), \dots, f(b)$ , with  $f$  a function,  $a$  and  $b$  numbers. If  $a > b$ , then the result is 0.

**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  $\text{eps} = \mathbf{Eps}$  has been reached. You may pass another numeric value for  $\text{eps}$  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  $[a, b]$ .  $\text{eps}$  is the relative error requested excluding cancellation of significant digits, and by default is equal to  $1e-15$ . Specifically,  $\text{eps}$  means:  $(\text{absolute error}) / (\int_a^b |f(x)| dx)$ .

The return is 1) the approximation to the integral, or **fail** if evaluation failed, and 2) an estimate  $\text{err}$  of the absolute error, where

- $\text{err} \geq 0$ : normal termination,
- $\text{err} < 0$ : abnormal termination, i.e. an convergent error has been detected: 1)  $f(x)$  or  $\frac{d^n}{dx^n} f(x)$  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  $\text{eps}$ . 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  $[a, \infty]$ , with  $a$  a number, using Double Exponential (DE) Transformation, also known as Tanh-sinh quadrature.

$f$  needs to be analytic over  $[a, \infty]$ .  $eps$  is the relative error requested excluding cancellation of significant digits, and by default is equal to  $1e-15$ . Specifically,  $eps$  means:  $(\text{absolute error}) / (\int_a^b |f(x)| dx)$ .

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** ( $f, a, [, \text{omega} [, \text{eps}]$ )

Integrates the oscillatory function  $f$  on the interval  $[a, \infty]$ , with  $a$  a number, using Double Exponential (DE) Transformation, also known as Tanh-sinh quadrature.

$f$  needs to be analytic over  $[a, \infty]$ .  $\text{omega}$  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:  $(\text{absolute error}) / (\int_a^b |f(x)| dx)$ .

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** ( $f, a, b [, \text{omega} [, \text{eps}]$ )

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  $\text{omega}$  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** ( $tp$ )

Computes a Newton interpolating polynomial and returns it as a univariate function. The interpolation points are passed in a table  $tp$ , with each point being represented as a 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
```

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

**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`  $\leq$  `Eps`, then an accuracy of  $1e-5$  is used for a second test. If it succeeds,  $x$  is included into both the first and the second sequence, indicating to the user that the first test failed.

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

See also: **calc.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`  $\leq$  `Eps`, then an accuracy of  $1e-5$  is used for a second test. If it succeeds,  $x$  is included into both the first and the second sequence, indicating to the user that the first test failed.

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

See also: **calc.maximum**.

**calc.polygen** ( $c_n, c_{n-1}, \dots, c_2, c_1$ )

Creates a polynomial  $p(x) = c_n \cdot x^{n-1} + c_{n-1} \cdot x^{n-2} + \dots + c_2 \cdot x + c_1$  from the coefficients  $c_n, c_{n-1}, \dots, c_2, c_1$  and returns it as a new function  $p := \langle\langle x \rangle\rangle p(x) \rangle\rangle$ , 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  $[a, a+step]$ ,  $[a+step, a+2 \cdot step]$ , ...,  $[b-step, b]$ , 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.



## 7.16 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.

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 v0.3.1 as of August 01, 2009 for Agena 0.21+

LUdecomp, add, backsubs, checkmatrix, checksquare, checkvector, coldim,
column, crossprod, det, diagonal, dim, dotprod, hilbert, identity, inverse,
isAntisymmetric, isDiagonal, isIdentity, isMatrix, isSquare, isSymmetric,
isVector, matrix, mmap, mmul, rowdim, scalarmul, sub, 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  $Ax = b$ . 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:

**$s1 \pm s2$**

Adds two vectors or matrices  $s1$ ,  $s2$ . The return is a new vector or matrix. This operation is done by applying the `__add` metamethod.

**$s1 - s2$**

Subtracts two vectors or matrices  $s1$ ,  $s2$ . The return is a new vector or matrix. This operation is done by applying the `__sub` metamethod.

**`k * s`**

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  $A \cdot x = b$ , 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)**

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  $n \times n$  Hilbert matrix  $H$ , with  $H[i][j] := 1/(i+j-x)$ . If  $x$  is not specified, then  $x$  is 1.

**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 matrix *A* of dimension *n*. 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.

**linalg.matrix** (*obj1*, *obj2*, ..., *objn*)

Creates a matrix from the given structures *obj<sub>k</sub>*. The structures are considered to be row vectors. Valid structures are vectors created with **linalg.vector**, tables, or sequences.

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.

**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 the 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.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.scalar mul** (*v* , *n*)

Performs a scalar multiplication by multiplying each element in vector *v* with the number *n*. The result is a new vector.

**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  $m \times n$ -matrix  $A$  and thus returns an  $n \times m$ -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  $a_1$ ,  $a_2$ , etc. The function also accepts a table or sequence of elements  $a_1$ ,  $a_2$ , 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.

**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 **map**. If the function has more than one argument, then all arguments *except the first* are passed right after the name of the vector.

Examples:

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

> vmap(<< (x, y) -> x > y >>, vector(1, 0, 1), 0): # 0 for y
[ true, false, true ]
```

See also: **linalg.vzip**, **linalg.mmap**, **linalg.mzip**.

**linalg.vzip (f, v1, v2 [, ...])**

This function zips together two vectors by applying the function  $f$  to each of its respective components. The result is a new vector  $v'$  where each element  $v'[k]$  is determined by  $v'[k] := f(v1[k], v2[k])$ .

$v1$  and  $v2$  must have the same dimension. The third to last argument to  $f$  must be given right after  $v2$ .

See also: **linalg.vmap**, **linalg.vzip**, **linalg.mmap**.

**linalg.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  $n$ , enter: **linalg.vector(n, [])**.



## 7.17 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.

A time value is always defined using the **clock.tm** constructor. You may apply the ordinary +, -, and \* operators in order to add, subtract or multiply values.

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 v0.1 as of March 30, 2009

add, adjust, mul, sub, tm
```

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 (s1, s2 [, ...])
```

The function adds two or more values of type `time`. The return is a value of type `time`.

```
clock.adjust (s)
```

The function adjusts the representation of time values in a time object `s` by applying the rules described in the description of `clock.time`.

```
clock.mul (x1, x2)
```

multiplies the numeric value `x1` with the time value `x2` (of type `time`). `mul` converts `x2` to seconds, and then multiplies `x2` with `x1`. The arguments may be in reverse order.

The return is a value of type `time`.

```
clock.sub (s1, s2 [, ...])
```

The function subtracts two or more values of type `time`. The return is a value of type `time`.

```
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 `time` of the form `tm(0, min, 0)`.

In the second form, both minutes and seconds are defined. The return is a value of type `time` 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 `time` 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.

If `_clockAdjust` is set `false` then no adjustments are made to the arguments. You can use `clock.adjust` to apply the adjustments described above.

## 7.18 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.

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 <b>ads.sizeof</b> 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]]]])
```

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.

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 is created. The string 'seq' creates a sequence. If the type passed is not known, <b>fail</b> 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, this option is ignored. If the number of columns is non-positive, <b>fail</b> is returned and no base is 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 are allowed (including the \0 character). If the string is too long, it is 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 there 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 safe, 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 safe, 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 by `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**.

## 7.19 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.

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.

### 7.19.1 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);
```

### 7.19.2 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);
```

### 7.19.3 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 set with **gdi.ink**.

```
> cyan := gdi.ink(d, .1, .5, .5);
> gdi.rectanglefilled(d, 200, 200, 400, 400, cyan);
```

To set a default colour for all subsequent drawings, use **gdi.useink**.

### 7.19.4 Closing a File or Window

To finally close the window, use **gdi.close**.

```
> gdi.close(d);
```

### 7.19.5 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.

### 7.19.6 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**.

### 7.19.7 Plotting geometric Objects easily

Like **gdi.plotfn**, the gdi function **plot** outputs geometric objects in a standard coordinate system where the point [0, 0] is in the 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 draws axes, uses a user-given 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.

The following geometric objects can be drawn with **gdi.plot**:

arcs	ellipses	rectangles
filled arcs	filled ellipses	filled rectangles
circles	lines	triangles
filled circles	points	filled triangles

A PLOT data structure is a sequence with the user-defined type 'PLOT'. It contains the name of the object, its attributes and colour. A line stretching from [0, 0] to [1, 1] in gray 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])
```

PLOT structures 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** 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

### 7.19.8 Colours, Part 2

The following colour names (of type string) are built in and are accepted by the **plot** and **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'.
```

### 7.19.9 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], in device d. A colour (an integer), 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], in device d. The arc is filled with either the default colour, or the one given by colour (an integer).

**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 in device *d*. *c* must be a number determined by **gdi.ink**. 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*, in device *d*. A *colour* (an integer), may be given optionally.

**gdi.circlefilled (d, x, y, r [, colour])**

Draws a filled circle around the centre *[x, y]* with radius *r*, in device *d*. The circle is filled with either the default colour, or the one given by *colour* (an integer).

**gdi.clearpalette (d)**

Removes all inks in 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 in 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*, in device *d*. A *colour* (an integer), 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*, in device *d*. The ellipse is filled with either the default colour, or the one given by *colour* (an integer).

**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 (s, o)**

Iterates a table, set, or sequence of pairs *s* and returns true if at least one of the left-hand sides of a pair in *s* is equal to *o*.

See also: **gdi.options**.

**gdi.initpalette (d)**

Allocates basic colours in device *d*.

**gdi.ink (d, r, g, b)**

Returns a palette colour value - an integer - for the colour given by its RGB values *r* (red), *g* (green), and *b* (blue), for device *d*. *r*, *g*, and *b* must be numbers *x* with  $0 \leq x \leq 1$ . The palette colour value can be given as an optional argument in most of the **gdi** functions, or be used in the **gdi.useink** function. Subsequent calls with the same arguments return different palette values.

**gdi.lastaccessed ()**

Returns the id (a number) of the last accessed device.

**gdi.line (d, x1, y1, x2, y2 [, colour])**

Draws a line from the first point [*x1*, *y1*] to the second point [*x2*, *y2*] in device *d*. A *colour* (an integer), 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'
'axescolour'	defines the colour of the axes (a colour string, see Chapter 7.19.6)	'axescolour':'cyan'
'bgcolour'	sets the background colour (a colour string, see Chapter 7.19.6)	'bgcolour':'yellow'
'colour'	sets the colour (a string, see Chapter 7.19.6) for the line to be plotted.	'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 <b>true</b> )	'labels':false
'labelsize'	sets the font size (a positive number) for axis labels ( <code>gdi.plotfn</code> function only)	'labelsize':6
'linestyle'	sets the dash style (a positive number) for the graph to be plotted ( <code>gdi.plotfn</code> 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 <b>false</b> .	'mouse':true

Option (key)	Meaning (value)	Example
'res'	resolution of the window or image file (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 ( <b>gdi.plotfn</b> function only)	'thickness':2
'title'	sets the title (a string) for the plot ( <b>gdi.plotfn</b> function only)	'title': 'Graph of sin(x)'
'titlecolour'	sets the colour (a string, see Chapter 7.19.6) of the title ( <b>gdi.plotfn</b> function only)	'titlecolour':'red'
'titlesize'	sets the font size (a positive number) of the title ( <b>gdi.plotfn</b> function only)	'titlesize':15
'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] in device d. A `colour` (an integer), may be given optionally.

**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 *fn* of functions, the graphs of the functions are plotted in one device - to a file or window.

If the `file` option is missing, the graphs are plotted to the a window (UNIX 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 graph. 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 be 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)
```

**gdi.rectangle (d, x1, y1, x2, y2 [, colour])**

Draws a rectangle with the lower left and upper right corners  $[x_1, y_1]$  and  $[x_2, y_2]$  in device  $d$ . A **colour** (an integer), 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  $[x_1, y_1]$  and  $[x_2, y_2]$  in device  $d$ . The rectangle is filled with either the default colour, or the one given by **colour** (an integer).

**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, in device  $d$ .

**gdi.setarc (s, x, y, r1, r2, a1, a2 [, colour])**

Inserts an arc around the centre  $[x, y]$  with  $x$  radius  $r_1$ ,  $y$  radius  $r_2$ , and the starting and ending angles  $a_1, a_2$ , 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  $r_1$ ,  $y$  radius  $r_2$ , and the starting and ending angles  $a_1, a_2$ , 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) 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  $s$ .

**gdi.setinfo** expects the x and y values in the following form:

```
'xdim':xmin:xmax, 'ydim':ymin:ymax,
```

where 'xdim', 'ydim' are the respective strings and  $xmin$ ,  $xmax$ ,  $ymin$ , and  $ymax$  represent numbers.

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);
```

```
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.plotfn** function.

For a list of valid plotting options, see **gdi.options**.

Internally, the function assigns the given options to the global environment variable `_Env.GdiDefaultOptions` which is checked by **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 in 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*], in device *d*. A *colour* (an integer), may be given optionally for the letters.

See also: **gdi.fontsize**.

**gdi.thickness (d, t)**

Sets the default thickness for all lines to *t* pixels, in 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*] in device *d*. A *colour* (an integer), 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*] in device *d*. The triangle is filled with either the default colour, or the one given by *colour* (an integer).

**gdi.useink (d, c)**

Sets the default colour *c* (a number) for all subsequent drawings, in device *d*. *c* must be a number determined by **gdi.ink**.



## 7.20 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.

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
<b>mapm.xabs</b>	absolute value	<b>mapm.xfactorial</b>	factorial
<b>mapm.xarccos</b>	arc cosine	<b>mapm.xdiv</b>	integer division
<b>mapm.xarccosh</b>	inverse hyperbolic cosine	<b>mapm.xln</b>	natural logarithm
<b>mapm.xadd</b>	addition	<b>mapm.xlog10</b>	common logarithm
<b>mapm.xarcsin</b>	inverse sine	<b>mapm.xmul</b>	multiplication
<b>mapm.xarcsinh</b>	inverse hyperbolic sine	<b>mapm.xpow</b>	power
<b>mapm.xarctan</b>	inverse tangent	<b>mapm.xsign</b>	sign
<b>mapm.xarctan2(x, y)</b>	4 quadrant inverse tangent	<b>mapm.xsin</b>	sine
<b>mapm.xarctanh</b>	hyperbolic inverse tangent	<b>mapm.xsincos</b>	sine and cosine
<b>mapm.xcbrt</b>	cubic root	<b>mapm.xsinh</b>	hyperbolic sine
<b>mapm.xcos</b>	cosine	<b>mapm.xsqrt</b>	square root
<b>mapm.xcosh</b>	hyperbolic cosine	<b>mapm.xsub</b>	subtraction
<b>mapm.xdiv</b>	division	<b>mapm.xtan</b>	tangent
<b>mapm.xexp</b>	exponential function	<b>mapm.xtanh</b>	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
<b>mapm.xceil</b>	ceil function	<b>mapm.xexponent</b>	exponent
<b>mapm.xfloor</b>	floor function	<b>mapm.xinv</b>	reciprocal
<b>mapm.xiseven</b>	test for even number	<b>mapm.xisint</b>	check for an integral
<b>mapm.xisodd</b>	test for odd number	<b>mapm.xmod</b>	modulus
<b>mapm.xround</b>	rounds downwards to the nearest integer	<b>mapm.xneg</b>	negates a number
<b>mapm.xcompare(x, y)</b>	comparison, returns -1 if $x < y$ , 0 if $x = y$ , and 1 if $x > y$	<b>mapm.xnumber</b>	converts an Agena number or a string representing a number to an arbitrary precision number
<b>mapm.xdigits</b>	sets the number of digits used in all subsequent calculations. With no argument, returns the current setting	<b>mapm.xtoNumber</b>	converts an arbitrary precision number to an Agena number
<b>mapm.xdigitsin</b>	significant digits	<b>mapm.xtoString</b>	converts an arbitrary precision number to a string

## 7.21 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.

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 like **fractals.flip**,
3. **fractals.draw** to draw fractals using escape-time iteration functions.

See Chapter 7.21.4 for some examples.

### 7.21.1 Escape-time Iteration Functions

**fractals.emarkmandel (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  $\lambda * \text{fractals.bea}(z)$ , 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  $\lambda * \text{fractals.cosxx}(z)$ , 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  $|z^3 - 1| < \text{radius}$  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  $\lambda * \text{fractals.bea}(z)$ , 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.emarkmandel**, but implemented in C.

**fractals.newton (x, y, iter, radius)**

Like **fractals.anewton**, but implemented in C.

## 7.21.2 Auxiliary Mathematical Functions

**fractals.bea (z)**

Takes the complex number  $z = x!y$  and returns the complex number  $\sin(x)*\sinh(y)+!*\cos(x)*\cosh(y)$ . 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  $\cos(x)*\cosh(y)+!*\sin(x)*\sinh(y)$ . It represents FRACTINT's buggy cos function till v1.6. This function may be mathematically meaningless, but it creates beautiful fractals.

**fractals.flip (z)**

Takes the complex number  $z$  and returns the complex number  $\text{imag}(z)!*\text{real}(z)$ .

### 7.21.3 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_center, y_center, x_width [, options])`**

Draws a fractal given by one of the escape-time iterator functions `iterator` with image centre `[x_center, y_center]` and of the total length on the x-axis `x_width`. `x_center` and `y_center` are numbers whereas `x_width` is a positive number.

Options are:

Option	Meaning	Example
<code>colour ~ f</code>	a colouring function $f$ of the form $f := \langle \langle x \rightarrow r, g, b \rangle \rangle$ . Predefined functions are: red, blue, violet, cyan, cyannew.	<code>colour ~ &lt;&lt; x -&gt; 0, 0, 0.05*x &gt;&gt;</code> <code>colour ~ blue</code>
<code>file ~ 'filename.suf'</code>	creates a GIF, PNG, or JPEG file, if the file suffix is .gif, .png, or .jpg	<code>file ~ 'mandel.gif'</code>
<code>iter ~ n</code>	maximum number of iterations with <code>n</code> a positive number; default is 128	<code>iter ~ 512</code>
<code>lambda ~ p</code>	lambda value <code>p</code> , a complex number, for fractals.[a]* functions like <b>albea</b>	<code>lambda ~ 1!0.4</code>
<code>map ~ 'filename.map'</code>	<p>FRACTINT colour map to be used to draw the fractal.</p> <p>The FRACTINT maps can be downloaded separately from:  <a href="http://agena.sourceforge.net/downloads.html#fractintmaps">http://agena.sourceforge.net/downloads.html#fractintmaps</a></p> <p>Put these files into the share folder of your Agena distribution, preserving the subfolder fractint. A valid path may thus be: <code>/usr/adena/share/fractint</code>.</p> <p>Alternatively, set the environment variable <code>_Env.FractintColorMaps</code> to the folder where your map files reside.</p>	<code>map ~ 'basic.map'</code>
<code>mouse ~ bool</code>	display pointer co-ordinates on console after image has been finished, if <code>bool = true</code> . Default: <code>bool = false</code> . <b>Click the right mouse button to quit printing co-ordinates.</b>	<code>mouse ~ true</code>

Option	Meaning	Example
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 nonnegative number: determines the number of rows after an image is being flushed to a file or window during computation	

Notes on the **update** option:

In Sun x86 Solaris and Linux, by default the image is updated each 10th row, in all other operating systems, including Sun Sparc Solaris, the default is 1. This behaviour in Sun x86 Solaris and Linux can be switched off by setting the global environment variable `_Env.FractOptimised` to **false** or **null**.

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.

## 7.21.4 Examples

```
> with 'fractals';
> 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);
> draw(fractals.lbea, 0, 0, 4, radius~128, iter~255, lambda~1.0!0.1);
```

There are further examples at the bottom of the `fractals.agn` file residing in the main Agena library folder.

## 7.22 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** functions.

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', zahl=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.

**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, Agenda 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, desc1 [, codepage] [, desc2, ..., desck])
```

creates a new xBase file with the file name `filename`.

`desck` are  $k$  fields (columns) the xBase file will have. `codepage` indicates the code page to be used (see below)<sup>14</sup>.

`desck` must be a pair of the following form:

#### 1. `field_name : data_type`

where `field_name` is a string and the name of the field to be added, and `data_type` is one of the strings '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].

---

<sup>14</sup> 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 may have occurred.



## Chapter Eight

# C API Functions





## 8 C API Functions

As already noted in Chapter 1, Agena features almost 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.

The following C API functions have been changed to remove automatic string-to-number conversion:

API function	Lua source file
lua_isnumber	lapi.c
lua_isstring	lapi.c
luaL_checknumber	lauxlib.c
luaL_checkinteger	lauxlib.c

Table 18: Modified Lua C API functions

Except for the above mentioned functions, no other modifications have been made to C API functions that are part of Lua 5.1.

For a description of the API functions taken from Lua, see its Lua 5.1 manual.

Agena features a macro **agn\_Complex** which is a shortcut for complex double.

The following API functions have been removed:

- lua\_dump

The following API functions have been added (see files `lapi.c` and `lua.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\_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.

### **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\_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\_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\_deletetable

```
LUA_API void agn_deletetable (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, 1) != 0) {
    /* 'key' is at index -4, 'value' at -3, function at -2, and 'value'
       at -1 */
    lua_call(L, 1, 1); /* call the function with one arg & one result */
    lua_pop(L, 1);      /* removes result of lua_call;
                        keeps 'key' for next iteration */
}
```

While traversing a table, do not call **lua\_tolstring** directly on a key, unless you know that the key is actually a string. Recall that **lua\_tolstring** changes the value at the given index; this confuses the next call to **lua\_next**.

## agn\_getbitwise

```
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\_getenv

```
LUA_API void agn_getenv (lua_State *L, const char *field)
```

Returns an entry from the global **\_Env** variable, a table containing various settings and information needed in the Agena environment. The function is equivalent to the call **\_Env.field** and puts the result on top of the stack.

## 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 if the value at **idx** is no function at all.

## 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 rtable), 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\_getseqstring**

```
const char *agn_getseqstring (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\_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\_getutype

```
int agn_getutype (lua_State *L, int idx);
```

Returns the user-defined type of a procedure, sequence, set, or pair at stack position `idx` as a string and pushes it onto the top of the stack. If no user-defined type has been defined, the function returns 0 and pushes nothing onto the stack.

See also: **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\_issequitype

```
int *agn_issequitype (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\_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\_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 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\_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\_pairgeti

```
void agn_pairgeti (lua_State *L, int idx, int n);
```

Returns the left operand of a pair at stack index `idx` if `n` is 1, and the right operand if `n` is 2, and puts it onto the top of the stack. You have to make sure that `n` is either 1 or 2.

## agn\_pairrawget

```
void agn_pairrawget (lua_State *L, int idx);
```

Pushes onto the stack the left or the right hand value of a pair `t`, where `t` is the value at the given valid index `idx` and the number `k` (`k=1` for the left hand side, `k=2` for the right hand side) is the value at the top of the stack. It does not invoke any metamethods. This function pops both `k` from the stack.

### **agn\_pairrawset**

```
void agn_pairrawset (lua_State *L, int idx);
```

Does the equivalent to  $p[k] := v$ , where  $p$  is a pair at the given valid index  $idx$ ,  $v$  is the value at the top of the stack, and  $k$  is the value just below the top.

This function pops both the key and the value from the stack. It does not invoke any metamethods.

### **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\_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 Agenda 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 roundoff 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, 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**

```
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.

### agn\_usedbytes

```
LUAU_UMEM agn_usedbytes (lua_State *L)
```

Returns the number of bytes used by the interpreter.

### 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);
```

Does the equivalent of `t[n] = string`, where `t` is the table at the given valid index `idx`, `n` is an integer, `string` 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);
```

Does the equivalent of  $t[n] = \text{num}$ , where  $t$  is the value at the given valid index  $\text{idx}$ ,  $n$  is an integer, and  $\text{num}$  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);
```

Does the equivalent of  $t[n] = \text{str}$ , where  $t$  is the value at the given valid index  $\text{idx}$ ,  $n$  is an integer, and  $\text{str}$  a string.

This function leaves the stack unchanged. The assignment is raw; that is, it does not invoke metamethods.

### **lua\_rawsetstringlint**

```
void lua_rawsetstringlint (lua_State *L, int idx, const char *str,  
                           int len, int n);
```

Does the equivalent of  $t[\text{str}] = n$ , where  $t$  is the value at the given valid index  $\text{idx}$ ,  $\text{str}$  a string,  $\text{len}$  the length of  $\text{str}$ , 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);
```

Does the equivalent of  $t[\text{str}] = n$ , where  $t$  is the value at the given valid index  $\text{idx}$ ,  $\text{str}$  a string, and  $n$  a Lua number.

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.

## 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 is popped from the stack.

## lua\_seqnext

```
int lua_seqnext (lua_State *L, int index);
```

Pops a key from the stack, and pushes the next key~value pair from the sequence at the given index. 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 index);
```

Pushes onto the stack the sequence value  $t[k]$ , where  $t$  is the sequence at the given valid index `index` 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 index, size_t n);
```

Pushes onto the stack the sequence value  $t[n]$ , where  $t$  is the sequence at the given valid index `index`.

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 index);
```

Pushes onto the stack the sequence value  $t[k]$ , where  $t$  is the sequence at the given valid index `index` 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 index);
```

Does the equivalent to  $s[k] := v$ , where  $s$  is a sequence at the given valid index `index`,  $v$  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);
```

Does the equivalent of `s[n] = string`, where `s` is the sequence at the given valid index `idx`, `n` is an integer, `string` 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 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.

## lua\_seqsetinumber

```
void lua_seqsetinumber (lua_State *L, int idx, int n, lua_Number num);
```

Sets the given Agena number `num` to index `n` of the sequence at stack index `idx`.

## lua\_seqsetistring

```
void lua_seqsetistring (lua_State *L, int idx, int n, const char *str);
```

Sets the given string `str` to index `n` of the sequence at stack index `idx`.

### lua\_sinsert

```
void lua_sinsert (lua_State *L, int idx);
```

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);
```

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);
```

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);
```

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 the key from the stack putting the Boolean value **true** or **false** in its place.

The function 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 not a boolean or is **false**, the function returns 0.

If the value at `idx` is **false**, the function returns -1.

If the value at `idx` is **true**, 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\_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\_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), \ (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.

Table 19: Metamethods taken from Lua

The '\_\_len' metamethod in Lua 5.1 to determine the size of an object was replaced with the '\_\_size' metamethod.

The following methods are new in Agena:

Index to metatable	Meaning
'abs'	<b>abs</b> operator
'arctan'	<b>arctan</b> operator
'cos'	<b>cos</b> operator
'eeq'	strict equality operator (==)
'entier'	<b>entier</b> operator
'even'	<b>even</b> operator
'exp'	<b>exp</b> operator
'finite'	<b>finite</b> operator
'lngamma'	<b>lngamma</b> operator
'in'	in binary operator (for tables and sequences only)
'int'	<b>int</b> operator
'intdiv'	integer division
'ipow'	exponentiation with an integer power
'ln'	<b>ln</b> operator
'__qsadd'	<b>qsadd</b> operator for table or sequence based user-defined types
'__sadd'	<b>sadd</b> operator for table or sequence based user-defined types
'sign'	<b>sign</b> operator
'size'	<b>size</b> operator
'sin'	<b>sin</b> operator
'sqrt'	<b>sqrt</b> operator
'tan'	<b>tan</b> operator
'__writeindex'	Procedure invoked when a value shall to be written to a table, set, sequence, or pair.

Table 20: Metamethods introduced with Agena

### A3 System Variables

Agena lets you configure the following settings:

System variable	Meaning
<b>homedir</b>	The path to the user's home directory
<b>mainlibname</b>	The path to the main Agena directory
<b>libname</b>	The paths to Agena libraries
<b>_Env.BufferSize</b>	The default buffer size for file operations for the <b>os.fcopy</b> and <b>binio.readlines</b> functions. It is equal to the C constant BUFSIZ in stdio.h.
<b>_Env.GdiDefaultOptions</b>	A table with all default plotting options for some functions in the <b>gdi</b> package. This table is set by <b>gdi.setoptions</b> .
<b>_Env.MaxLong</b>	The maximum integral value of the C type int32_t on your platform; do not change this value.

System variable	Meaning
<code>_Env.MinLong</code>	The minimum integral value of the C type <code>int32_t</code> on your platform; do not change this value.
<code>_Env.More</code>	The number of entries in tables and sets printed by <b>print</b> and the end-colon functionality before issuing the <code>`press any key`</code> prompt. Default is 40.
<code>_Env.PathSep</code>	The token that separates paths in <code>libname</code> ; by default is <code>'/'</code> . Do not change this value as it is used by the <b>with</b> function.
<code>_Env.Release</code>	A sequence containing the string <code>`AGENA`</code> , the main interpreter version as a number, the subversion as a number, and the patch level as a number, as well.
<code>_Env.WithProtected</code>	A set of names (passed as strings) that cannot be overwritten by the <b>with</b> function.
<code>_Env.WithVerbose</code>	If set to <b>false</b> , the <b>with</b> function will not display warnings, the initialisation string, and the short names assigned. Default is <b>true</b> .
<code>PROMPT</code>	Defines the prompt Agena displays at the console.
<code>_RELEASE</code>	Release information on the installed Agena release, returned as a string, e.g. <code>'AGENA &gt;&gt; 0.90.0'</code> .

Table 21: System variables

All `_Env*` settings are reset by the **restart** statement to their original defaults, whereas those settings the user defines with the **kernel** function will never be modified or deleted by a **restart**.

Some of the default settings can be found at the bottom of the `lib/library.agn` file.

See also:

- Chapter 7.1 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]);

try arg1 :: string;
try arg2 :: 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
-b	print compilation time of Agena binary with startup message
-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
-n	do not run initialisation file `agenda.ini`
-p path	sets <path> to <b>libname</b> , 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
--	stop handling options
-	execute stdin and stop handling 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 `_EnvPrint` function or settings defined in the `lib/library.agn` file. You may change these functions or settings according to your needs.

Table index	Type	Functionality
<code>_EnvPrint.Table</code>	function	defines how to print a table, overriding the built-in default
<code>_EnvPrint.LongTable</code>	function	defines how to print a table if <code>_Env.LongTable</code> has been set true
<code>_EnvPrint.Set</code>	function	defines how to print a set, overriding the built-in default
<code>_EnvPrint.Sequence</code>	function	defines how to print a sequence, overriding the built-in default
<code>_EnvPrint.Pair</code>	function	defines how to print a pair, overriding the built-in default
<code>_EnvPrint.Complex</code>	function	defines how to print a complex value, overriding the built-in default

Table index	Type	Functionality
<code>_EnvPrint.EmptyLine</code>	boolean	If set to <b>true</b> , a newline is printed at the console after entering a statement (and pressing the RETURN key) and before the result appears on screen. Default: unassigned, i.e. no newline.
<code>_EnvPrint.EncloseStrings</code>	string	if set, Agena outputs strings with the prepending and appending string assigned to <code>_EnvPrint.EncloseStrings</code>
<code>_EnvPrint.Procedure</code>	function	defines how to print a procedure, overriding the built-in default
<code>_EnvPrint.ZeroedCmplxVals</code>	boolean	When set to <b>true</b> , 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 <b>null</b> .

Alternative `_EnvPrint` functions might look like the following one:

```
> _EnvPrint.Set := proc(s) is
>   write('set(');
>   if size s > 0 then
>     for i in s do
>       write(i, ', ');
>     od;
>     write('\b\b');
>   fi;
>   write(')');
> end;

> _EnvPrint.Complex := 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 **restarted**. 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 by 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	<code>&lt;drive:&gt;\WINNT\Profiles\&lt;username&gt;</code>
2000, XP, 2003	<code>&lt;drive:&gt;\Documents and Settings\&lt;username&gt;</code>
Vista and 7	<code>&lt;drive:&gt;\Users\&lt;username&gt;</code>

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');
```

## 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.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notices and this permission notice shall be included in all copies or portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.

IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

### B2 GNU GPL v2 Licence

The Solaris, Linux, Windows, OS/2, Mac OS X, and DOS binaries are distributed under the GNU GPL v2 licence reproduced below:

GNU GENERAL PUBLIC LICENSE  
Version 2, June 1991

Copyright (C) 1989, 1991 Free Software Foundation, Inc.,  
51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

## 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 noncommercial 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

```
* =====
* Copyright (C) 1993 by Sun Microsystems, Inc. All rights reserved.
*
* Developed at SunPro, a Sun Microsystems, Inc. business.
* Permission to use, copy, modify, and distribute this
* software is freely granted, provided that this notice
* is preserved.
* =====
```

### B4 GNU Lesser General Public License

Agenda uses the g2 graphic library which is distributed under the GNU LGPL v2.1 licence reproduced below:

#### GNU LESSER GENERAL PUBLIC LICENSE Version 2.1, February 1999

Copyright (C) 1991, 1999 Free Software Foundation, Inc.  
59 Temple Place, Suite 330, Boston, MA 02111-1307 USA  
Everyone is permitted to copy and distribute verbatim copies  
of this license document, but changing it is not allowed.

[This is the first released version of the Lesser GPL. It also counts as the successor of the GNU Library Public 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 Other Copyright remarks**

The Solaris, Linux, Mac OS X, and Windows binaries include code from the gd package which has been published with the following copyright notices:

Portions copyright 1994, 1995, 1996, 1997, 1998, 1999, 2000, 2001, 2002 by Cold Spring Harbor Laboratory. Funded under Grant P41-RR02188 by the National Institutes of Health.

Portions copyright 1996, 1997, 1998, 1999, 2000, 2001, 2002 by Boutell.Com, Inc.

Portions relating to GD2 format copyright 1999, 2000, 2001, 2002 Philip Warner.

Portions relating to PNG copyright 1999, 2000, 2001, 2002 Greg Roelofs.

Portions relating to gdttf.c copyright 1999, 2000, 2001, 2002 John Ellson (ellson@lucent.com).

Portions relating to gdft.c copyright 2001, 2002 John Ellson (ellson@lucent.com).

Portions copyright 2000, 2001, 2002, 2003, 2004, 2005, 2006, 2007 Pierre-Alain Joye (pierre@libgd.org).

Portions relating to JPEG and to color quantization copyright 2000, 2001, 2002, Doug Becker and copyright (C) 1994, 1995, 1996, 1997, 1998, 1999, 2000, 2001, 2002, Thomas G. Lane. This software is based in part on the work of the Independent JPEG Group. See the file README-JPEG.TXT for more information.

Portions relating to WBMP copyright 2000, 2001, 2002 Maurice Szmurlo and Johan Van den Brande.

Permission has been granted to copy, distribute and modify gd in any context without fee, including a commercial application, provided that this notice is present in user-accessible supporting documentation.

This does not affect your ownership of the derived work itself, and the intent is to assure proper credit for the authors of gd, not to interfere with your productive use of gd. If you have questions, ask. "Derived works" includes all programs that utilize the library. Credit must be given in user-accessible documentation.

This software is provided "AS IS." The copyright holders disclaim all warranties, either express or implied, including but not limited to implied warranties of merchantability and fitness for a particular purpose, with respect to this code and accompanying documentation.

Although their code does not appear in gd, the authors wish to thank David Koblas, David Rowley, and Hutchison Avenue Software Corporation for their prior contributions.

### **B6 MAPM Copyright Remark (Mike's Arbitrary Precision Math Library)**

Copyright (C) 1999 - 2007 Michael C. Ring

This software is Freeware.

Permission to use, copy, and distribute this software and its documentation for any purpose with or without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation.

Permission to modify the software is granted. Permission to distribute the modified code is granted. Modifications are to be distributed by using the file 'license.txt' as a template to modify the file header. 'license.txt' is available in the official MAPM distribution.

To distribute modified source code, insert the file 'license.txt' at the top of all modified source code files and edit accordingly.

This software is provided "as is" without express or implied warranty.