agena >>

A Programming Language

Primer and Reference
for Version 0.13

by Alexander Walz
February 23, 2009

Contact: In case you find bugs, errors in this manual, have proposals, or questions regarding Agena, please contact the author at: agena.info@t-online.de

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

Credits

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. 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.

Table of Contents

Chapter One

Introduction

1 Introduction

1.1 Features

Agena is an easy-to-learn interpreted programming language suited for sophisticated procedural programming.

It combines features of Lua 5, Algol 60, Algol 68, Maple, ABC, ANSI C, and Sinclair ZX Spectrum BASIC.

While Agena's syntax looks a lot like Algol 68, its implementation is based on the original Lua 5.1 sources created by Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes.

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

- assignments,
- loops,
- conditions,
- procedures, and
- package handling.

Besides providing these basic operations, it has extended programming features described later in this manual, such as

- high-performance processing of complex 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,
- easy-to-use package initialisation functions,
- 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 values known from mathematics. With all of these types, you can easily build fast applications.

1.2 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 where start, stop, and step values are optional,
- combined numerical **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,
- **try/else** data type validation.

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),
- remember tables for conducting recursion at high speed and at low memory consumption,
- 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:

- functions to support fast text processing (see **in**, **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 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.

Agena is shipped with 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 your programs; additional packages for basic statistical functions, Linear Algebra, undergraduate Calculus, a library to create and read binary files and a text-based database are available, as well. 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 packages you have already written for Lua without modifying its code in 99.9 % of all cases.

1.3 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.

Study of many books and websites on various programming languages such as Algol 60, Algol 68, and ABC along with Maple and my various ideas on the `perfect` language helped to conceive a completely new Algol 68-syntax based language with high-speed general 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 and NT 4 using the MinGW GCC compiler. Further programming has been done on a Sun Sparc Ultra 5 and a Sun Blade 150 running Solaris 10 and on openSuSE Linux 10.3 to make the interpreter work in UNIX environments.

Chapter Two

Installing & Running Agena

2 Installing and Running Agena

2.1 Solaris

In Solaris, put the gzipped Agena package into any directory. Assuming you install Agena on a Sparc, uncompress the package by entering:

```
> gzip -d agena-0.13.0-sparc-local.gz
```

Then install it with the Solaris package manager:

```
> pkgadd -d agena-0.13.0-sparc-local
```

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

Make sure you have the ncurses and readline libraries installed. From the command line type **agena** and press RETURN.

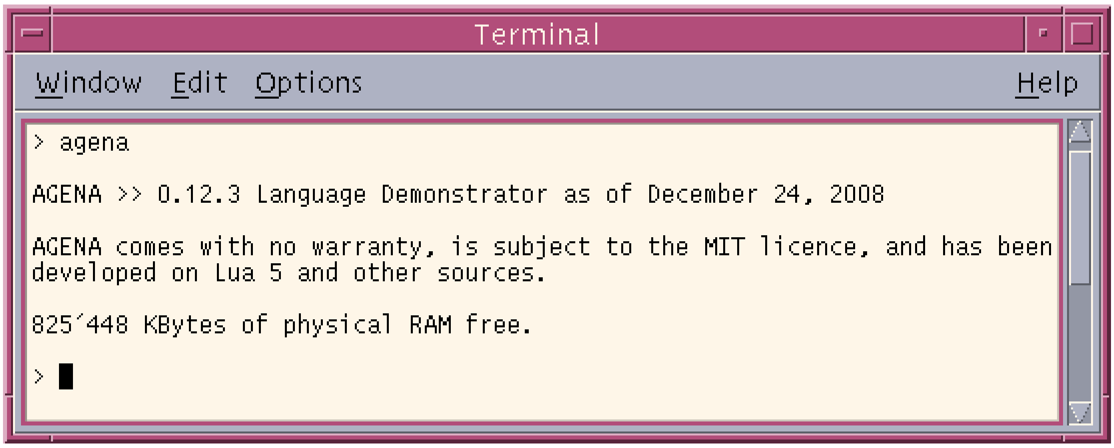

Image 1: Startup message in Solaris

The procedure for Solaris for x86 CPUs is the same.

2.2 Linux

In Linux, put the Agena rpm package into any directory and install it by typing:

```
> rpm -ivh agena-0.13.0-linux-i386.rpm
```

This installs the executable into the `/usr/local/bin` folder and the rest of all files into `/usr/agena`. The `/usr/agena` directory is called the `main Agena folder`. Note that you must have the *ncurses* and *readline* libraries installed before.

From the command line type **agena** and press RETURN.

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 folder (the default) or set it later manually in the Windows Control Panel, icon `System`.

2.4 Agena Initialisation

When you start Agena, the following actions are taken:

1. The package tables for the libraries shipped with the standard edition of Agena (e.g. math, strings, etc.) are created so that those package procedures are 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 Agena system variable **_EnvAgenaPath** pointing to the main Agena folder is set by either querying the environment variable AGENAPATH or - if not set - checking whether the current working directory contains the string '/agena' and building the path accordingly. In UNIX, if the path could not be determined as described before, _EnvAgenaPath is by default set to `/usr/agena`, but in Windows there is no such fallback. The variable is used extensively in the **with** and **readlib** functions. If it could not be set these two functions do not work, but all others do.

4. The standard Agena library `library.agn` in the `/agena/lib` folder is loaded and run. The `library.agn` file includes functions written in the Agena language which complement the C libraries. If the standard library does not exist, this step is skipped without any errors.

5. An initialisation file - if present - called `agena.ini` residing in the `/agena/lib` folder is loaded and run. As with `library.agn`, this file contains code written in the Agena language that you may customise with pre-set variables, auxiliary procedures, etc. that shall be available in every Agena session. If the initialisation file does not exist, no error is issued, and the Agena session begins.

Chapter Three

Overview

3 Overview

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

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 the 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.

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

3.2 Getting familiar

Assume you would like 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
```

If you would like to compute the sum of 1 to 10, and hold the result in a variable called `r`, input:

```
> r := 0;

> for i from 1 to 10 do
>    r := r + i
> od;


> r:
55
```

There are many functions available in various libraries. To compute the arc sine, use the **arcsin** function in the **math** package;

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

You can easily write your own functions, for example one called `deg` that converts radians to degrees.

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

To compute the value of the function at Pi/4, just input:

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

Try one of the built-in standard operators. **lower** converts all letters from upper case to lower case.

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

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 the current contents:

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

To delete entries, just type:

```
> birthdates['Morpheus'] := null

> birthdates:
Trinity ~ 1970/12/24
Neo ~ 1970/01/01
```

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

```
> ans:
agena
```

The console screen can be cleared in both the Win32 and UNIX versions by just entering the keyword **cls**:

```
> cls
```

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

```
> restart;
```

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

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

You may put this statement into the `agena.ini` file in the Agena `lib` folder, if you do not want to manually change the prompt every time you start Agena.

3.3 Comments

You should always document the code you have written so that others or even yourself will understand its meaning later.

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

```
> # this is a single-line comment

> a := 1; # a contains a number
```

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

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

Now let us learn more about Agena.

[1] Multi-line comments cannot begin in the very first line of a program file. Use a single comment 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 **undefined** and **infinity**
string	any text
boolean	Booleans (e.g. **true**, **false**, and **fail**)
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 **null** in sequential order
pair	a pair of two values of any type
complex	a complex number consisting of a real and an imaginary number

Table 1: 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, so no declarations of variable names are needed.

Valid names	Invalid names
`var`	`1var`
`_var`	`1_`
`var1`	
`_var1n`	
`_1`	
`ValueOne`	
`valueTwo`	

Table 2: Names

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

```
abs add and arctan as assigned break by bye case char clear cls copy cos
dec delete dict do elif else end entier enum esac even exp external
fail false fi filled finite for from gammaln global if imag in
inc insert int intersect into is isnull join keys left ln local lower
minus nargs not null od of or proc qadd real replace restart return
right seq shift si sign sin size skip split sqrt subset tan then to
trim true try type union unique upper utype while xsubset

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

The following symbols denote other tokens:

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

4.2 Assignment

Values can be assigned to names in the following fashions:

name := *value*
$name_1$, $name_2$, ..., $name_k$:= $value_1$, $value_2$, ..., $value_k$
$name_1$, $name_2$, ..., $name_k$ -> *value*

In the first form, one value is stored in one variable, whereas in the second form, called \`multiple assignment statement\`, $name_1$ is set to $value_1$, $name_2$ is assigned $value_2$, etc. In the third form, called \`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 *name*$_1$ [, *name*$_2$, ...]
enum *name*$_1$ [, *name*$_2$, ...] **from** *value*

In the first form, *name*$_1$, *name*$_2$, etc. are enumerated starting with the numeric value 1.

```
> enum ONE, TWO;

> ONE:
1

> TWO:
2
```

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

```
> enum THREE, FOUR from 3

> THREE:
3

> FOUR:
4
```

4.4 Deletion

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.

4.5 Precedence

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

```
or
and
< > <= >= = <>
in subset xsubset union minus intersect
..
+ - split
* / % \ shift
:
not -(unary)
^ **
!
```

As usual, you can use parentheses to change the precedence of an expression. The concatenation (`..`), exponentiation (`^`, `*`) and pair (`:`) operators are right associative. 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.

```
> 1e10:
10000000000

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

If a number ends with the letter `K`, `M`, or `d`, then the number is multiplied with 1,024, 1,048,576 (1,024^2), or 12, respectively. If a number ends with the letter `k` or `m`, then the number is multiplied with 1,000 or 1,000,000, respectively.

```
> 2k:
2000

> 1M:
1048576

> 12d:
144
```

4.6.2 Arithmetic Operations

Agena has the following arithmetical operators:

Operator	Operation	Details / Example
+	Addition	`1 + 2 » 3`
-	Subtraction	`3 - 2 » 1`
*	Multiplication	`2 * 3 » 6`
/	Division	`4 / 2 » 2`
^	Exponentiation with rational power	`2 ^ 3 » 8`
**	Exponentiation with integer power	faster than `^`, `2 ** 3 » 8`
%	Modulus	`5 % 2 » 1`
\	Integer division	`5 \ 2 » 2`
shift	Bitwise shift	If the right-hand side is positive, the bits are shifted to the left (multiplication with 2), else they are shifted to the right (division by 2).

Table 3: Arithmetic

Agena has a lot of mathematical functions both built into the kernel and available in the **math**, **stats**, linalg, and **calc** libraries. Table 3 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[2].

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

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

[2] Check the **with** function which provides an easy way to define short names for package procedures.

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

Table 4: Common mathematical functions

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[3] which are also faster:

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

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

```
> inc a;

> a:
3

> dec a;

> a:
2

> inc a, 2;

> a;
4
```

[3] Finishing an **inc** or **dec** statement with a colon instead of a semicolon does not work.

```
> dec a, 3;

> a:
1
```

4.6.4 Mathematical Constants

Agena features the following arithmetic constants:

Constant	Meaning
degrees	Factor 1/Pi*180 to convert radians to degrees
_EnvEps	Equals 1.4901161193847656e-08
Exp	Constant e = exp(1) = 2.71828182845904523536
I	Imaginary unit
infinity	Infinity
Pi	Constant pi = 3.14159265358979323846
radians	Factor Pi/180 to convert degrees to radians
undefined	An expression stating that it is undefined, e.g. a singularity.

Table 5: 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 letter **I**. Most of Agena's mathematical operators and functions know how to handle complex numbers.

```
> 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`, `arctan`, `cos`, `exp`, `ln`, `sign`, `sin`, `sqrt`, `tan`, and unary minus. With these operators, you can also mix numbers and complex numbers in expressions. You will find that most functions of the **math** package are also applicable to complex values.

Complex values are of type **complex**.

4.7 Strings

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

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

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.

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$(1);
I
```

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

stringname$(*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):
i
```

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 quote into the string, put a backslash right in front of it:

```
> '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 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[4]:

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

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

```
> `text:
text
```

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

```
> `/proglang/agena/utils/utils.agn:
/proglang/agena/utils /utils.agn
```

Agena has basic operators useful for text processing:

Operator	Return	Function
s **in** t	number or **null**	Checks whether a substring s is included in string t. If true, the position of the first occurrence of s in t is returned; otherwise **null** is returned.
replace(s, p, r)	string	Replaces all patterns p in string s with string r. If p is not in s, then s is returned unchanged.
s **split** d	table of strings	Splits a string into items with d as the delimiting character. The items are returned in a sequence of strings.
size(s)	number	Returns the length of string s. If s is the empty string, 0 is returned.
abs(s)	number	Returns the numeric ASCII code of character s.
char(n)	string	Returns the character corresponding to the given numeric ASCII code n.
lower(s)	string	Converts a string to lowercase. Western European diacritics are recognised.
upper(s)	string	Converts a string to uppercase. Western European diacritics are recognised.
trim(s)	string	Deletes leading and trailing spaces as well as excess embedded spaces.

Table 6: 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 which are separated by white spaces:

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

str is eight characters long:

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

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

```
> abs(str$(1)):
97
```

translated back to

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

Put all characters in str to uppercase:

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

And now the reverse:

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

The **replace** functionality easily replaces all occurrences of a substring with another one:

```
> replace(str, 'string', 'text'):
a text
```

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 program which statements to execute and thus which statements not to execute.

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

- relational operators (>, <, =, <=, >=, <>, **in**, **subset**),
- logical operators (**and**, **or**, **not**),
- logical names: **true**, **false**, **fail**, and **null**.

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`
<>	not equals	`1 <> 2`

Table 7: Relational operators

Logical operators are:

Operator	Description	Examples

`and`	Both operands must evaluate to **true** so that the Boolean expression results to **true**. Otherwise the result is **false**.	`true and true » true`
		`false and false » false`
		`true and false » false`
		`false and true » false`

`or`	At least one of the operands must evaluate to **true** so that the Boolean expression results to **true**. If neither of the operands is true, the expression is **false**.	`true or true » true`
		`true or false » true`
		`false or true » true`
		`false or false » false`

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

Table 8: 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.

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. **fail** <> **false results to** true.

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

4.9 Tables

Tables are used to represent any complex data structure. 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 from Viking Lander 1 which landed 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 input:

```
> VL1[1.02]:
1 ~ 7.71
2 ~ -78.28
```

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

Stripped down versions of tables are sets and sequences which are described later. You will find that most operations on tables introduced in this chapter are also applicable to sets and sequences.

4.9.1 Arrays

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

[[*value*$_1$ [, *value*$_2$, ...]] **]**

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

The numbers 1, 2, and 3 are the *keys* or *indices* of table A. The corresponding table *values* are 4, 5, and 6. 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 VL in the last chapter is not.

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

tablename[*key*]

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

If a table contains other tables, you may get their values by passing the respective keys in consecutive order:

tablename[key_1][key_2][...]

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

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

```
> A[1]:
1 ~ 3
2 ~ 4
```

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

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

Tables may be nested:

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

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

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

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

create **table** $name_1$ [, **table** $name_2$, ...]

$name_1$:= *[ ]*

```
> create table B;
```

creates the empty table B,

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

does exactly the same.

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

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

> B:
1 ~ a
```

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

insert *value*$_1$ [, *value*$_2$, ...] **into** *name*

```
> insert 'b' into B;

> B:
1 ~ a
2 ~ 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 *value*$_1$ [, *value*$_2$, ...] **from** *name*

```
> insert 'b' into B;

> delete 'b' from B;

> B:
empty table
```

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 size 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 size 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 default size will be exceeded.

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

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

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 with it. 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.

```
> table a(5);

> table a, 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*$_1$ ~ *value*$_1$ [, *key*$_2$ ~ *value*$_2$, ...]]]

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

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

Here is another example with strings as keys:

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

> dic:
mickey ~ mouse
donald ~ duck
```

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

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

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

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

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

Further entries can be added with assignments such as:

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

which is the same as typing

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

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

create dict *name*$_1$(*size*$_1$) [, **dict** *name*$_2$(*size*$_2$), ...]

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

create {**table** | **dict**} *name*$_1$[(*size*$_1$)] [, {**table** | **dict**} *name*$_2$[(*size*$_2$)], ...]

4.9.3 Table, Set and Sequence Operators

Agena features some built-in table, set and sequence operators which are shown in Table 6. A `structure` in this context is a table, set, or sequence.

Operator	Return	Function
c **in** A	Boolean	Checks whether the structure A contains the given value c.
filled A	Boolean	Determines whether a structure contains at least one value. If so, it returns **true**, else **false**.
A = B	Boolean	Checks whether two tables A, B, or two sets A, B, or two sequences A, B contain the same values regardless of the number of their occurence; if B is a reference to A, then the result is also **true**.

Operator	Return	Function
A <> B	Boolean	Checks whether two sets/tables/sequences A, B do not contain the same values regardless of the number of their occurence; if B is a reference to A, then the result is **false**.
A **subset** B	Boolean	Checks whether the values in structure A are also values in B regardless of the number of their occurence. The operator also returns **true** if A = B.
A **xsubset** B	Boolean	Checks whether the values in structure A are also values in B. Contrary to **subset**, the operator returns **false** if A = B.
A **union** B	table, set, seq	Concatenates two tables, or two sets, or two sequences A, B simply by copying all its elements - even if they occur multiple times - to a new structure. With sets, all items in the resulting set will be unique, i.e. they will not appear multiple times.
A **intersect** B	table, set, seq	Returns all values in two tables, two sets, or two sequences A, B that are included both in A and in B as a new structure.
A **minus** B	table, set, seq	Returns all the values in A that are not in B as a new structure.
copy A	table, set, seq	Creates a deep copy of the structure A, i.e. if A includes other tables, sets, or sequences, copies of these structures are built, too.
join A	string	Concatenates all strings in the table or sequence A.
size A	number	Returns the size of a table A, i.e. the actual number of key~value pairs in A. With sets and sequences, the number of items is returned.
sort A	table, seq	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.
unique A	table, seq	Removes multiple occurrences of the same value and returns the result in a new structure. With tables, also removes all holes (`missing keys`) by reshuffling its elements. This operator is not applicable to sets, since they are already unique.
add A	number	Sums up all numeric table or sequence values. If the table or sequence is empty or contains no numeric values, **null** is returned. Sets are not supported.
qadd A	number	Raises each value in a table or sequence to the power of 2 and sums up these powers. If the table or sequence is empty or contains no numeric values, **null** is returned. Sets are not supported.

Table 9: Table, set, or sequence and set operators

Here are some examples - try them with sets 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 included in 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 in 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']
```

returns [1 ~ 'a', 2 ~ 'b'].

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

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

returns **true**.

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

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

4.9.4 Table Functions

Agena has a number of functions to work on tables only. The most basic are:

Function	Description	Further detail
tables.put(*t*, *key*, *value*)	Inserts index *key* with value *value* to table t.	It shifts up the original element at position *key* and all other elements to the right.
tables.remove(*t*, *key*)	Removes index *key* and its corresponding value from *t*.	All elements to the right are shifted down, so that no holes are created.

Table 10: Basic table procedures

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 ~ 1
2 ~ 2
3 ~ 3
```

also yields:

```
> B:
1 ~ 1
2 ~ 2
3 ~ 3
```

Use **copy** to create a copy of the contents of a table. If the table contains tables, copies of these tables are also made (so-called `deep copies`).

```
> B := copy(A);

> insert 4 into A;

> B:
1 ~ 1
2 ~ 2
3 ~ 3
```

Thus with structures such as tables, sets, pairs, or sequences, all names at the left of an `->` operator will point to the very same structure at the right. This behaviour may be changed in future versions of Agena.

```
> A, B -> []

> A[1] := 1

> B:
1 ~ 1
```

4.10 Sets

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

{ [*item₁* [, *item₂*, ...]] }

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 list 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[*key*]

If an item is stored to a set, Agena returns **true**:

```
> colours['red']:
true
```

If an item is not in the given set, the return is **false**.

```
> colours['yellow']:
false
```

If you want to add or delete items to or from a set, use the **insert** and **delete** statements. The standard assignment statement `setname[key] := value` is not supported with sets.

insert *item*$_1$ [, *item*$_2$, ...] **into** *name*
delete *item*$_1$ [, *item*$_2$, ...] **from** *name*

```
> insert 'yellow' into colours;
```

The **in** operator checks whether an item is part of a set - it is an alternative to the indexing method explained above, and returns **true** or **false**, too.

```
> 'yellow' in colours:
true
```

The data type of a set is **set**.

```
> type(colours):
set
```

You may predefine sets with a given number of entries according to the following syntax:

create set *name*$_1$(*size*$_1$) [, **set** *name*$_2$(*size*$_2$), ...]

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 number of entries as explained above. More items than stated at initialisation can be entered anytime, since Agena automatically enlarges the respective set accordingly.

Sets are useful in situations where the number of occurrences of a specific item does not matter. 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.

As with tables, sets support metamethods which you can define to extend the functionality of Agena operators. Metamethods will be explained later in Chapter 6.

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.

Sequences store items in sequential order. Like in tables, an item may be included multiple times. Sequences are indexed with positive integers in the same fashion as table arrays are, starting at index 1.

Metamethods for operator overloading that allows to extend the functionality of the built-in Agena operators to sequences are supported, too (see Chapter 6.9 for more details). A sequence may hold no, one or more items.

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.

Suppose we want to define a pair of two values. You may enter these values into the sequence using the **seq** operator.

seq([*item*$_1$ [, *item*$_2$, ...]] **)**

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

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

You may access the items the usual way:

seqname[*numeric_key*]

```
> a[1]:
0

> a[2]:
1
```

If the index is larger than the current size of the sequence, an error is returned[5].

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

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

The way Agena outputs sequences can be changed by using the **settype** function.

```
> settype(a, 'pair');

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

The **gettype** function returns the new type you defined above as a string:

```
> gettype(a):
pair
```

If no user-defined type has been set, **gettype** returns **null**.

Once the type of a sequence has been set, the **utype** function also returns this user-defined sequence type instead of `'sequence'`.

```
> utype(a):
pair
```

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);

> utype(a):
sequence
```

The **create seq** statement allows to create an empty sequence and define an empty sequence with enough memory allocated to hold the given number of elements which can be inserted later in a session. Agena automatically extends the sequence, if the predetermined number of items is exceeded.

create seq $name_1$ [, **seq** $name_2$, ...] **create seq** $name_1(size_1)$ [, **seq** $name_2(size_2)$, ...]

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

```
> 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, since `holes` in a sequence are not allowed. The next free position in a is at index 3, however a larger index will be 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 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

> delete 2, 3 from a

> a:
empty sequence
```

If you assign a sequence to a variable, only a reference to the sequence is stored in the variable. Thus sequences behave the same like tables and sets, i.e. in a statement like `A := seq(); B := A`, A and B point to the same sequence.

```
> A := seq()

> B := A

> A[1] := 10

> B:
seq(10)
```

The following operators, functions, and statements work on sequences:

Function	Description	Example
=	Equality check	`a = b`
<>	Inequality check	`a <> b`
insert	Inserts one or more elements	`insert 1 into a`
delete	Deletes one or more elements	`delete 0, 1 from a`
copy	Creates an exact copy of a sequence; deep copying is supported so that sequences inside sequences are properly treated.	`b := copy a`
filled	Checks whether a sequence has at least one item	`filled a`
in	Checks whether an element is stored in the sequence, returns **true** or **false**.	`0 in seq(1, 0)`
join	Concatenates all strings in a sequence in sequential order.	`join(a)`
size	Returns the current number of items	`size a`
sort	Sorts a sequence in place.	`sort(a)`

Function	Description	Example
type	Returns the general type of a sequence, i.e. **sequence**.	`type a`
utype	Returns the user-defined type of a sequence, or the basic type if no special type has been defined.	`utype a`
unique	Reduces multiple occurrences of an item in a sequence to just one.	`unique a`
unpack	Unpacks a sequence. See **unpack** in Chapter 7.1.	`unpack(a)`
settype	Sets a user-defined type for a sequence	`settype(a, 'duo')`
gettype	Returns a user-defined type for a sequence	`gettype(a)`
setmeta	Assigns a metatable to a sequence.	`setmeta(a, mtbl)`
getmeta	Returns the metatable stored to a sequence.	`getmeta(a)`

Table 11: Basic sequence procedures

4.12 More on the create statement

With the **create** statement, you cannot only initialise any number of tables, 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:
[]      []      {}      seq()   []
```

4.13 Pairs

The structure which holds exactly two values of any type (including **null**) is the *pair*. A pair cannot hold less or more values, but its values can be changed. Conceived originally to allow passing options as function arguments in a more flexible way, it is defined with the colon operator:

$item_1$: $item_2$

```
> p := 1:2

> p:
1:2
```

The **left** and **right** operators provide the only read access to its left and right operands; the standard indexing method using indexed names is not supported:

left [(] *pair* [)]
right [(] *pair* [)]

```
> left(p):
1

> right(p):
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.

```
> utype(p):
pair

> settype(p, 'duo');

> p:
duo(2, 3)

> utype(p):
duo

> gettype(p):
duo
```

The only other operators besides **left** and **right** that work on pairs are equality, inequality (= and <>), **type**, **utype**, and **in**.

```
> p = 3:2:
false
```

With pairs consisting of numbers, the **in** operator checks whether a left-argument number is part of a closed numeric interval given by the given right-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.

Summary:

Function	Description	Example
=	Equality check	`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 **true** or **false**. If at least one value x, a, b is not a number, the operator returns **fail**.	`1.5 in 1:2`
left	Returns the left operand of a pair.	`left(a)`
right	Returns the right operand of a pair.	`right(a)`
type	With pairs, always returns `'pair'`.	`type(a)`
utype	Returns either the user-defined type of the pair, or the basic type (`'pair'`) if no special type was defined for the pair.	`utype(a)`
settype	Sets a user-defined type for a pair.	`settype(a, 'duo')`
gettype	Returns the user-defined type of a pair.	`gettype(a)`
setmeta	Sets a metatable to a pair.	`setmeta(p, mtbl)`
getmeta	Returns the metatable stored to a pair.	`getmeta(p)`

Table 12: 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 *condition*$_1$ **then**
  statements$_1$
[**elif** *condition*$_2$ **then**
  statements$_2$]
[**else**
  statements$_3$]
fi

The condition must always evaluate to one of the Boolean values **true**, **false**, or **fail**, **to null**, a number, or a string.

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 a condition results to **true**, a number (including 0), or a string, its corresponding **then**-clause is executed. If none of the above mentioned conditions is **true**, the **else** clause is executed if present, otherwise Agena proceeds with the next statement following the **if** statement.

This also means that if a condition is or results to **null** or **fail**, the **then**-clause is never run.

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 **false** is not true, so nothing is printed:

```
> if 1 <> 1 then
>    print('this will never be printed')
> fi;
```

An **if** statement with an **else** clause:

```
> if false then
>    print('this will never be printed')
> else
>    print('this will always be printed')
> fi;
this will always be printed
```

An **if** statement with an **elif** clause:

```
> if 1 = 2 then
>    print('this will never be printed')
> elif 1 < 2 then
>    print('this will always be printed')
> fi;
this will always be printed
```

An **if** statement with **elif** and **else** clauses:

```
> if 1 = 2 then
>    print('this will never be printed')
> elif 1 < 2 then
>    print('this will always be printed')
> else
>    print('neither will this be printed')
> fi;
this will always be printed
```

5.1.2 is-Operator

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

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

This means that the result is *expression*$_1$ if *condition* is **true**, a number (including 0), or a string, and *expression*$_2$ otherwise.

Example:

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

which is the same as:

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

The **is** operator only evaluates the expression that it will return. Thus the other expression which will not be returned will never be checked for semantic correctness, e.g. out-of-range string indices, etc. You may nest **is** operators.

5.1.3 case Statement

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

case *name*
  of $value_{11}$ [, $value_{12}$] **then** $statements_1$
  [**of** $value_{21}$ [, $value_{22}$] **then** $statements_2$]
  [**of** ...]
  [**else** $statements_k$]
esac

```
> a := 'k';

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

> result:
consonant
```

You can add as many **of** .. **then** statements as you like. Fall through is not supported. This means that if one **then** clause is executed, Agena will not evaluate the following **of** clauses and will proceed with the statement right after the closing **esac** keyword.

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, it iterates the loop body again and again as long as the condition remains true. If the condition is **false**, no further iteration is done and control returns to the statement following right after the loop body.

If the condition is **false** 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 $+1$. If stop is not given, the last iteration value is **infinity**[6].

It then checks whether *start* $<=$ *stop*. If so, it executes *statements* and returns to the top of the loop, increments *name* by *step* and then checks whether the new value is less or equal *stop*. If so, *statements* are executed again. If *step* is not given, the control variable is always incremented by $+1$.

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

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

The 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 infinity while math.fact(i) < 1k do od
```

[6] These loops do not run infinitely, but stop at the numeric value of the C constant HUGE_VAL which varies among systems.

```
> 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[7] within the loop, or if a for/while loop is terminated because the **while** condition evaluated to **false**, then the control variable is set to the loop's last iteration value before quitting the loop. There will be no increment with the loop's step size.

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

```
> for i from 2 to 1 by -1 do
>    print(i)
> od
2
1
```

A special form is the **to** .. **do** loop which does not feature a control variable and iterates exactly n times.

```
> to 2 do
>    print('iterating')
> od
iterating
iterating
```

5.2.3 for/in Loops for Tables

are used to traverse tables[8], strings, sets, and sequences. 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]
```

[7] See chapter 5.2.7 for more information in the **break** statement.

[8] To be more general, for/in loops iterate over functions called iterators. Check out the Lua documentation for more information.

```
> 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 table if you want to change, add, or delete its entries.

5.2.4 for/in Loops for Sequences

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

5.2.5 for/in Loops for 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** *str* **do** *statements* **od**
for *value* **in** *str* **do** *statements* od
for keys *value* **in** *str* **do** *statements* **od**

The following code converts a word to a sequence of abstract vowel, ligature, and consonant placeholders 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 for 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/while Loops

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

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** *str* **while** *condition* **do** *statements* **od**
for keys *key* **in** *str* **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 *str*.

5.2.8 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. Programs are usually represented as procedures. The words `procedure` and `function` are used synonymously.

6.1 Procedures

In general, procedures cluster a sequence of statements into abstract units which then can be repeatedly invoked.

Writing procedures in Agena is quite simple:

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

All the values that a procedure shall process are given as *parameters par*$_1$, etc. A function may have no, one, or more parameters.

A procedure usually uses local variables which are private to the procedure and cannot be used by other procedures or in the Agena interactive level.

Global variables are supported in Agena, as well. All values assigned in the interactive level are global, and you can 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** token which may be put anywhere in the procedure body.

return *value* [, *value2*, ...]

The following procedure computes the factorial of an integer[9]:

```
> 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;
```

[9] The library function **math.fact** is much faster.

It is called using the following syntax:

funcname([*arg*$_1$ [, *arg*$_2$, ...]])

```
> 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 *name*$_1$ [, *name*$_2$, ...]
local *name*$_1$ [, *name*$_2$, ...] := *value*$_1$ [, *value*$_2$, ...]
local *name*$_1$ [, *name*$_2$, ...] -> *value*$_1$ [, *value*$_2$, ...]
local table *name*$_1$ [, **table** *name*$_2$, ...]
local table *name*$_1$(*size*$_1$) [, **table** *name*$_2$(*size*$_2$), ...]
local set *name*$_1$ [, **set** *name*$_2$, ...]
local set *name*$_1$(*size*$_1$) [, set *name*$_2$(*size*$_2$), ...]
local seq *name*$_1$ [, **seq** *name*$_2$, ...]
local seq *name*$_1$(*size*$_1$) [, **seq** *name*$_2$(*size*$_2$), ...]
local enum *name*$_1$ [, *name*$_2$, ...] [**from** *value*]

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

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

In the fourth to ninth form, one or more local empty table(s), set(s), or sequence(s) called *name*$_1$, etc. are created.

In the last form, *name*$_1$, etc. are declared local with a subsequent enumeration of those names.

It is possible to mix declarations of names and on the same stroke create empty structures such like empty tables, sets, and sequences. For example, the statement

```
local a, table b, c, seq d, set e, seq f, table g(10)
```

declares the names `a` through `g` and also creates the empty tables `b` and `g`, the empty sequences `d` and `f`, and the empty set `e`.

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 within all procedures.

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 for 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 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** 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 non-**null** arguments *actually* passed in a procedure call by querying the system variable **nargs** inside the respective procedure.

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.

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

6.5 Passing Options

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 size varargs 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
```

6.6 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 features facilities for this: the **type** operator and the **try** statement. It 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
   in function fact, line 3
```

Another, more efficient way of type checking is provided by the **try** statement that is around twice as fast as the **if**/type/**error** combination.

try *name₁* [, *name₂*, ...] **as** *typename*
try *name₁* [, *name₂*, ...] **as** *typename* **else** *errorstring*

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

```
> fact := proc(n) is
>    try n as number;
>    if n < 0 then return null
>    elif n = 0 then return 1
>    else return fact(n-1)*n
>    fi
> end;

> fact('10'):
Error, line 2: expected number, got string for argument #1.
   in function fact, line 2

> fact := proc(n) is
>    try n as 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, line 2: for argument #1: bad value for argument
   in function fact, line 2
```

Note that opposed to the **type** operator, the **try** statement only checks for basic types, i.e. user-defined types for procedures, tables, sequences, and pairs are not recognised.

6.7 Shortcut Procedure Definition

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

<< ([*par₁* [, *par₂*, ...]]) -> expr >>

Let us define a simple factorial function.

```
> fact := << (x) -> exp(gammaln(x+1)) >>

> fact(4):
24
```

Brackets around the arguments are optional.

```
> 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.

6.8 User-Defined Procedure Types

The **settype** function allows to group procedures p1, p2, ..., by giving them a specific type (passed as a string) just as it does with sequences, tables, and pairs.

settype($proc_1$ [, $proc_2$, ...], 'your_proctype'**)**

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

utype($proc_1$**)**

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')

> utype(f):
constant

> type(f):
procedure
```

6.9 Scoping Rules

In Agena, variables live in blocks or `scopes`. A block may contain one or more other blocks. A 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 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** clause of an **if**-statement live only in this then part. The same applies to variables declared local in **elif** or **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 following example demonstrates how this 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
```

6.10 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-1 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 $z = z\hat{}2{+}c$. Since the loop control variable i has been declared external, it can be used in the **return** statement.

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
```

6.11 Packages

6.11.1 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) is
>    try n as number;
>    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;
```

Now we can use our new package.

```
> helpers.ndigits(0):
1

> helpers.ndigits(-10):
2

> helpers.ndigits(.1):
Error: argument is not an integer
   in function ndigits, line 4
```

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` of 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
```

6.11.2 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.

```
> restart;

> 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 code into the `helpers.agn` file, save the file and restart Agena:

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

> restart;

> use `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 any position:

```
> 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) is
   try n as number;
   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
```

6.12 Remember tables

Agena features remember tables which if present hold the results of previous calls to Agena procedures. If an Agena 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* for short. Note that C API functions are not supported, only functions written in the Agena language are.

The feature 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.

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

First the function is defined:

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

Only after the function has been created, the rtable (short for remember table) can be set up. The **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:

```
1 ~ [1~1]
2 ~ [1~1]
3 ~ [1~2]
4 ~ [1~3]
5 ~ [1~5]
6 ~ [1~8]
7 ~ [1~13]
8 ~ [1~21]
9 ~ [1~34]
...
```

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 a table, and the returns are passed in a separate 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
```

For completeness, all basic functions which work on rtables are the following:

Procedure	Details
rinit(f)	Initialises an rtable for function `f`.
hasrtable(f)	Checks whether procedure `f` possesses an rtable.
rget(f)	Returns the rtable of function `f`.
rset(f, argument, return) **rset**(f, [arguments], [returns])	Adds function argument(s) and the corresponding return(s) to the rtable of procedure `f`.
rdelete(f)	Deletes the rtable of function `f` entirely. If you want to use a new one, you have to initialise it with rinit again.

Table 13: Functions for remember tables

Please check Chapter 7.1 for more details on their use.

6.13 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[10].

This method of defining new 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 plus token, determining their absolute value, 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, b) is
>    try a, b as number;
>    create local seq r(2);
>    insert a into r;
>    insert 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:

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

```
> cmplx := proc(a, b) is
>    try a, b as number;
>    create local seq r(2);
>    insert a into r;
>    insert 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, line 1: attempt to perform arithmetic on a sequence value
```

Metamethods are defined with dictionaries, so 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 a table, set, sequence, or pair. See Appendix for a list of all available method names. To overload the plus operator use the `'__add'` name.

Assign this metamethod to any name.

```
> cmplx_mt := [
>    '__add' ~ proc(a, b) is
>                  return cmplx(a[1]+b[1], a[2]+b[2])
>               end
> ]
```

Next, we must assign this metatable to the cmplx structure with the **setmetatable** function. We have to extend the constructor by one line, the call to **setmetatable**:

```
> cmplx := proc(a, b) is
>    try a, b as number;
>    create local seq r(2);
>    insert a into r;
>    insert 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.

```
> cmplx_mt.__abs := << (a) -> math.hypot(a[1], a[2]) >>;
```

The metatable now contains two methods.

```
> cmplx_mt:
__add ~ procedure(003FE3E8)
__abs ~ procedure(0046CE80)


> z := cmplx(1, 1)

> abs(z):
1.4142135623731
```

It would be quite fine if complex values would be output the usual way using the standard x + y*i* notation. This can be done with the `'__tostring'` method which must return a string.

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

> z:
1+1i
```

To avoid using the **cmplx** constructor in calculations, we want to define the imaginary unit I = 0+i and use it in subsequent operations. Before assigning the I unit, we have to add a metamethod for multiplying a number with a complex number.

```
> cmplx_mt.__mul := proc(a, b) is
>    if utype(a) = 'cmplx' and utype(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 utype(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 utype(a) = 'cmplx' and utype(b) = 'cmplx' then
>       return cmplx(a[1]+b[1], a[2]+b[2])
>    elif type(a) = number and utype(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 very slow - way to use a notation like `z.re` and `z.im` in both read and write operations for variables is provided by the `'__index'` and `'__newindex'` metamethods, respectively.

The __index metamethod for reading values from a structure works as follows:

- If the object is a table, then the metamethod is called if the call to an indexed name results to **null**.
- If the object is a set, then the metamethod is called if the call to an indexed name results to **false**.
- If the object 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 __newindex metamethod to write values to a structure works as follows:

- If the object is a table, then the metamethod is called if the key to which a value shall be assigned is not already present in the structure.
- If the object is a set, sequence or pair, then the metamethod is always called.

The respective procedures assigned to the __index and __newindex 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 define the metamethods. Both will be capable to accept expressions like `a.re` and `a[1]`.

```
> 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;

> cmplx_mt.__newindex := proc(x, y, z) is  # write operation
>    if type(y) = string then
>       rawset(x, cmplx_indexing[y], z)
>    else
>       rawset(x, y, z)
>    fi
> end;
```

You can now use them.

```
> a:
1+2i

> a.re:
1

> a.im := 3

> a:
```

```
1+3i
```

6.14 File I/O

Agena 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** package.

6.14.1 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:/agena/lib/agena.ini') do
>    print(i, j)
> od
execute := os.execute;    null
getmeta := getmetatable;        null
setmeta := setmetatable;        null
```

6.14.2 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 Agena 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.');
```

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 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 5.1distribution provide useful functions that are implemented directly through the C API. Some of these functions provide essential services to the language (e.g., **next** and **getmetatable**; others provide access to "outside" services (e.g., I/O); and others could be implemented in Agena itself, but are quite useful or have critical performance requirements that deserve an implementation in C (e.g., **sort**).

The following text is based on Chapter 5 of the Lua 5.1 manual and includes all the new operators, functions, and packages provided by Agena.

Lua functions which were deleted from the code are not described. References to Lua were not deleted from the original text. If an explanation mentions Lua, then the description also applies to Agena.

All libraries are implemented through the official C API and are provided as separate C modules. Currently, Lua/Agena has the following standard libraries:

- basic library;
- package library;
- string manipulation;
- table manipulation;
- mathematical functions;
- input and output;
- operating system facilities;
- database operations;
- debug facilities.

Except for the basic and package libraries, each library provides all its functions as fields of a global table or as methods of its objects. Agena operators have directly built into the kernel (the Virtual Machine), so they are part of any library.

7.1 Basic Functions

The basic library provides some core functions to Lua. 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 returns the absolute value of x. Complex numbers are supported.

If x is a Boolean, it returns 1 for **true**, 0 for **false**, and -1 for **fail**.

If x is null, **abs** returns -2.

If x is a string of only one character, **abs** returns 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 (v)`

Returns all global names that are assigned values in the environment.

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

`approx (a, b [, eps])`

compares the two numbers `a` and `b` and checks whether they are approximately equal using a simplified relative approximation algorithm developed by Donald H. Knuth. If `eps` is omitted, **_EnvEps** is used. (The algorithm checks whether the relative error is bound to a given tolerance eps.)

The function returns **true** if `a` and `b` are considered equal or **false** otherwise.

`assigned (v)`

This Boolean operator checks whether any value different from **null** is assigned to the expression `v`. If `v` is a constant, i.e. a number or a string, the operator always returns **false**. If `v` *evaluates* to a constant, the operator returns **true**.

See also: **null**.

`assume (v [, message])`

Issues an error when the value of its argument v 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 (o)`

With the table o returns a new table with

- the current maximum number of key~value pairs allocable to the array and hash parts of `o`; 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 `o`; in the resulting table, these values are indexed with keys `'array_assigned'` and `'hash_assigned'`.

With the set o 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 `o`; in the resulting table, this value is indexed with the key `'hash_assigned'`.

With the sequence o 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 curent number of items actually assigned to `o`; in the resulting table, this value is indexed with the key `'size'`.

bye

Quits the Agena session. No argument 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 the values, too.

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 concat other values than strings, e.g.:

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

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)`. Lua itself does not use this variable; changing its value does not affect any environment, nor vice-versa. (Use **setfenv** to change environments.)

`filled (obj)`

This Boolean operator checks whether a table, set, or sequence `obj` contains at least one item and returns **true** if so; otherwise it returns **false**.

`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 Lua (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*.

(The original Lua function collectgarbage has been renamed gc in Agena.)

`getfenv (f)`

Returns the current environment in use by the function. `f` can be a Lua 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 a Lua function, or if f is 0, getfenv returns the global environment. The default for f is 1.

`globals (f)`

determines whether function f includes global variables (names which have not been defined local).

`getmeta (object)`

`getmetatable (object)`

If `object` does not have a metatable, returns **null**. Otherwise, if the object's metatable has a '__metatable' field, returns the associated value. Otherwise, returns the metatable of the given object.

gettype (o)

returns the type - set with **settype** - of a function, sequence, or pair o 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**.

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

hasrtable (f)

checks whether function `f` has a remember table. It returns **true** if it has got one, and **false** otherwise.

isnull (v)

This Boolean operator checks whether an expression `v` is unassigned, i.e. is **null**. If `v` is a constant, i.e. a number or a string, the operator always returns **false**.

See also: **assigned**.

left (p)

returns the left operand of a pair.

See also: **right**.

load (func [, chunkname])

Loads a chunk using function `func` to get its pieces. Each call to func 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 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.

loadfile ([filename])

Similar to load, but gets the chunk from file `filename` or from the 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

```
assert(loadstring(s))()
```

map (f, o [, ...])

This operator maps a function f or anonymous function to all the values in table, set, sequence, or pair o. The function must return only one value. The type of return is the same as of o. If o has metamethods, the return also has them. If o is a sequence or pair, its special type if present is copied, as well.

If function f has only one argument, then only the function and the structure o 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 table or set.

Examples:

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

> map(<< (x, y) -> x > y >>, [-1, 0, 1], 0):  # 0 for y
1 ~ false
2 ~ false
3 ~ true
```

See also: **zip**.

maptoset (function, obj [, ...])

maps a function or anonymous function 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 (t [, 'sorted'])

Returns the maximum of all numeric values in table or sequence `t`. If the option `'sorted'` is passed than the function assumes that all values in `t` are sorted in ascending order and returns the last entry.

See also: **min**.

min (t [, 'sorted'])

Returns the minimum of all numeric values in table or sequence `t`. If the option `'sorted'` is passed than the function assumes that all values in `t` are sorted in ascending order and returns the first entry.

See also: **max**.

next (o [, index])

Allows a program to traverse all fields of a table or all items of a set or sequence. 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.

op (index, ···)

If `index` is a number, returns all arguments after argument number `index`. Otherwise, `index` must be the string `'#'`, and `op` returns the total number of extra arguments it received. The function is useful for accessing multiple returns (e.g. `op(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.

print (···)

Receives any number of arguments, and prints their values to `stdout`, using the **toString** function to convert them to strings. **print** is not intended for formatted output, but only as a quick way to show a value, typically for debugging. For formatted output, use **strings.format** .

In Agena, **print** also prints the contents of tables and nested tables to stdout if no `__tostring` metamethods are assigned to them. The same applies to sets and sequences. After **_EnvMore** number of lines, **print** halts for the user to press any key for further output. Press 'q', 'Q', or the Escape key to quit. The default for **_EnvMore** is 40 lines, but you may change this value in the Agena session or via the `agena.ini` file.

rawequal (v1, v2)

Checks whether `v1` is equal to `v2`, 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 (set, 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 given `set`.

This function returns `obj` or `set`.

rdelete (func)

deletes the remember table of procedure `func` 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**, **debug.doubleendiantest**.

readlib (packagename)

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).
The function first tries to find the binary C library wich must reside in the /lib folder of the Agena directory. If it finds it, it loads and runs the library and proceeds with the next step.

Next, the function tries to locate an Agena text file library in the folder `/packagename` of the Agena directory and loads, runs it when found and quits thereafter. Otherwise it tries to find the library in the `/lib` folder in the Agena directory, loads and runs it when found.

Make sure that in your operating system, you have set the environment variable AGENAPATH to the main folder where Agena resides and that the path does not end with a slash. In Win32, you my set the variable with the following statement:

```
SET AGENAPATH=d:/agena
```

The function returns **true** if the package has been successfully loaded and executed, or **false** if an error occurred.

You may also pass a complete file name (with or without path) to the function. In this case the given file is loaded and executed.

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

register (pkgname, name1 [, name2, ...])

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 $name_k$ can be used as short names for package functions instead of the fully written function names.

So, instead of later calling a function by "pkgname.name(arguments)" you may use the shortcut "name(arguments)". See **with** for more details.

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

_RELEASE

A global variable (not a function) that holds a string containing the current interpreter version. The current contents of this variable is `'AGENA 0.12'`.

remember (func)
remember (func, tab)
remember (func, null)

administrates remember tables.

In the first form, the remember table stored to procedure `func` is returned. See **rget** for more information.

In the second form, **remember** adds arguments and returns to the remember table of function `func`. If the remember table of `func` has not been initialised before, **remember** creates it. If there are already values in the remember table, they are kept and not deleted.

If `func` has only one argument and one return, the function arguments and returns are passed as key~value pairs in table `tab`.

If `func` has more than one argument, the arguments are passed in a table. If `func` 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 `func` is destroyed and a garbage collection run to free up space occupied by the former rtable.

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

See chapter 6.8 for examples.

`remove (f, o [, ...])`

returns all values in table, set, or sequence o 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.

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 ~ 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 ~ 1
```

`restart`

Restarts an Agena session. No argument is needed.

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

If the Agena session is restarted with **restart**, all values in the Agena environment are unassigned including the environment variable **_G**, but except of **_origG and** _EnvAgenaPath. Then all entries in **_origG** are read and assigned to the new environment. After this, the library base file `agena.lib` and thereafter the initialization file `agena.ini` - if present - are read and executed. Finally, restart runs a garbage collection.

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

`rget (func)`

returns the contents of the current remember table of procedure `func`. The function actually returns the rtable as a reference so you can manipulate it from the outside:

```
> fib := proc(n) is
>    assume(n >= 0);
>    return fib(n-2) + fib(n-1)
> end;

> remember(fib, [0~0, 1~1]);

> f := rget(fib)

> f:
1 ~ [1~1]
```

```
0 ~ [1~0]

> f[2] := [1]

> f:
1 ~ [1~1]
2 ~ [1~1]
0 ~ [1~0]

> rget(fib):
1 ~ [1~1]
2 ~ [1~1]
0 ~ [1~0]
```

right (p)

returns the right operand of a pair.

See also: **left**.

rinit (func)

creates a remember table (an empty table) for procedure `func`. 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 `func`, it is overwritten. **rinit** returns **null**.

rset (func, arguments, returns)

The function adds one (and only one) function-argument-and-returns `pair` to the already existing remember table of procedure `func`. `arguments` must be a table array, `returns` must also be a table array.

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])`.

See chapter 6.8 for examples.

run (filename)

Opens the named file and executes its contents as a Lua 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). The original name of this function in Lua is dofile.

See also: **readlib**, **with**.

save (o, fn)

saves an object of any type `o` into a binary file denoted by file name `fn`.

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

See also: **read**, **debug.doubleendiantest**.

select (f, o [, ...])

returns all values in table, set, or sequence o that satisfy a condition determined by function f. The type of return is determined by the type of second argument.

If the function has only one argument, then only the function and the object are passed to **select**.

```
> select(<< x -> x > 1 >>, [1, 2, 3]):
2 ~ 2
3 ~ 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 of o to the new structure.

setfenv (f, table)

Sets the environment to be used by the given function. `f` can be a Lua 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.

setmeta (table, metatable)

setmetatable (table, metatable)

Sets the metatable for the given table. (You cannot change the metatable of other types from Lua, 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 table. Contrary to tables, sets do not have metatables.

settype (o [, ...], str)

settype (o [, ...], null)

In the first form the function sets the type of one or more procedures, sequences, tables, or pairs o to the name denoted by string str. **gettype** and **utype** will then return this string when called with o.

In the second form, by passing the **null** constant, the user-defined type is deleted, and **gettype** thus will return **null** whereas **utype** will return the basic type of o.

If o 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 (v)

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

With sets and sequences, the operator returns the number of items in v.

With strings, the operator returns the number of characters in string v, i.e. the length of v.

sort (o [, comp])

Sorts table or sequence elements in a given order, in-place, from o[1] to o[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 < 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.

time ()

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

toTable (s)

If s is a string, the function splits it into its characters, and returns them in a table with each character in s as a table value in the same order as the characters in s.

If s is a sequence or set, the function converts it into a table.

type (v)

This operator returns the basic type of its only argument, coded as a string. The possible results of this function are `'null'` (a string, not the value **null**), `'number'`, `'string'`, `'boolean'`, `'table'`, `'set'`, `'sequence'`, `'pair'`, `'complex'`, `'procedure'`, `'thread'`, and `'userdata'`.

If v is a sequence, pair, or procedure with a user-defined type, then **type** always returns the basic type, i.e. `'sequence'` or `'pair'`, or `'procedure'`, respectively.

See also: **utype**.

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

Returns the elements from the given table or sequence. This function is equivalent to

```
return list[i], list[i+1], ···, list[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 list, as defined by the length operator.

used ()

returns the total memory in use by Agena in Kbytes. It is a shortcut for `gc('count')`. The function is written in the Agena language and included in the `library.agn` file.

utype (v)

This operator returns the user-defined type - if it exists - of its only argument, coded as a string.

A special type can be defined for procedures, pairs, and sequences with the **settype** function. If there is no user-defined type for v, then the basic type is returned, i.e. `'null'` (a string, not the value **null**), `'number'`, `'string'`, `'boolean'`, `'table'`, `'set'`, `'sequence'`, `'pair'`, `'complex'`, `'procedure'`, `'thread'`, and `'userdata'`.

See also: **type**.

whereis (tbl, x)

returns the indices for a given value `x` in table `t` as a table. The function is written in the Agena language and included in the `library.agn` file.

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. In a first step, the function looks into the `lib` folder of the main Agena library to find the package files. If it did not find it in the `lib` folder, it switches to the `packagename` folder in the main Agena directory and tries to load it from there. Note that the package files must reside either in the `lib` or in the `packagename` folder.

- If either the Agena library or the C library could not be found, **with** proceeds without errors. If both are missing, an error is returned.

- 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 applying to 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.

  In Windows, make sure that you have set the environment variable AGENAPATH to the main folder where Agena resides and that the path does not end with a slash. You my set the variable with the following statement, e.g.:

  ```
  SET AGENAPATH=d:/agena
  ```

  if Agena is installed in the `d:\agena` folder. In UNIX, Agena by default searches in the `/usr/agena` folder if AGENAPATH has not been set.

  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**.

`write ([fh,] v1 [, v2 ...] [, delim ~ <str>])`

This function prints a sequence of values 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>` (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.write**.

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

This function prints a sequence of values 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 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 be table arrays, and not dictionaries.

If `seq1` or `seq2` have user-defined types or metatables, they are copied to the resulting structure, as well.

See also: **map**.

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 Lua 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 Lua 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 basic facilities for loading and building modules in Lua. It exports two of its functions directly in the global environment: **require** and **module**. Everything else is exported in a table `package`.

`module (name [, ···])`

Creates a module. If there is a table in `package.loaded[name]`, this table is the module. Otherwise, if there is a global table t with the given name, this table is the module. Otherwise creates a new table t and sets it as the value of the global name and the value of package.loaded[name]. This function also initialises t._NAME with the given name, t._M with the module (t itself), and t._PACKAGE with the package name (the full module name minus last component; see below). Finally, module sets t as the new environment of the current function and the new value of package.loaded[name], so that require returns t.

If name is a compound name (that is, one with components separated by dots), module creates (or reuses, if they already exist) tables for each component. For instance, if name is a.b.c, then module stores the module table in field c of field b of global a.

This function may receive optional options after the module name, where each option is a function to be applied over the module.

`require (modname)`

Loads the given module. The function starts by looking into the table **package.loaded** to determine whether `modname` is already loaded. If it is, then require returns the value stored at package.loaded[modname]. Otherwise, it tries to find a loader for the module.

To find a loader, first require queries package.preload[modname]. If it has a value, this value (which should be a function) is the loader. Otherwise require searches for a Lua loader using the path stored in package.path. If that also fails, it searches for a C loader using the path stored in package.cpath. If that also fails, it tries an all-in-one loader (see below).

When loading a C library, require first uses a dynamic link facility to link the application with the library. Then it tries to find a C function inside this library to be used as the loader. The name of this C function is the string 'luaopen_' concatenated with a copy of the module name where each dot is replaced by an underscore. Moreover, if the module name has a hyphen, its prefix up to (and including) the first hyphen is removed. For instance, if the module name is a.v1-b.c, the function name will be luaopen_b_c.

If require finds neither a Lua library nor a C library for a module, it calls the all-in-one loader. This loader searches the C path for a library for the root name of the given module. For instance, when requiring a.b.c, it will search for a C library for a. If found, it looks into it for an open function for the submodule; in our example, that would be luaopen_a_b_c. With this facility, a package can pack several C submodules into one single library, with each submodule keeping its original open function.

Once a loader is found, require calls the loader with a single argument, modname. If the loader returns any value, require assigns it to package.loaded[modname]. If the loader returns no value and has not assigned any value to package.loaded[modname], then require assigns **true** to this entry. In any case, require returns the final value of package.loaded[modname].

If there is any error loading or running the module, or if it cannot find any loader for the module, then require signals an error.

`package.cpath`

The path used by **require** to search for a C loader.

Lua initialises the C path **package.cpath** in the same way it initialises the Lua **path package.path**, using the environment variable `LUA_CPATH` (plus another default path defined in `luaconf.h`).

`package.loaded`

A table used by **require** to control which modules are already loaded. When you require a module `modname` and `package.loaded[modname]` is not **false**, require simply returns the value stored there.

`package.loadlib (libname, funcname)`

Dynamically links the host program with the C library `libname` . Inside this library, looks for a function funcname and returns this function as a C function. (So, funcname must follow the protocol (see lua_CFunction)).

This is a low-level function. It completely bypasses the package and module system. Unlike require, it does not perform any path searching and does not automatically adds extensions. libname must be the complete file name of the C library, including if necessary a path and extension. funcname must be the exact name exported by the C library (which may depend on the C compiler and linker used).

This function is not supported by ANSI C. As such, it is only available on some platforms (Windows, Linux, Mac OS X, Solaris, BSD, plus other Unix systems that support the dlfcn standard).

package.path

The path used by **require** to search for a Lua loader.

At start-up, Lua initialises this variable with the value of the environment variable `LUA_PATH` or with a default path defined in `luaconf.h`, if the environment variable is not defined. Any ';;' in the value of the environment variable is replaced by the default path.

A path is a sequence of templates separated by semicolons. For each template, require will change each interrogation mark in the template by filename, which is modname with each dot replaced by a "directory separator" (such as "/" in Unix); then it will try to load the resulting file name. So, for instance, if the Lua path is

```
'./?.agn;./?.lc;/usr/local/?/init.agn'
```

the search for a Lua loader for module foo will try to load the files ./foo.agn, ./foo.lc, and /usr/local/foo/init.agn, in that order.

package.preload

A table to store loaders for specific modules (see **require**).

package.seeall (module)

Sets a metatable for `module` with its __index field referring to the global environment, so that this module inherits values from the global environment. To be used as an option to function module.

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	è	È	é	É	Ë						
ï	Ï	î	Î	ì	Ì	í	Í	ý	Ý	ÿ			
ô	Ô	ö	Ö	ò	Ò	ø	Ø	ó	Ó	õ	Õ		
û	Û	ù	Ù	ü	Ü	ú	Ú						
ç	Ç	ñ	Ñ	ð	Ð	þ	Þ	ß					

7.4.1 Kernel Operators and B asic Library Functions

replace (s1, s2, s3)
replace (s1, struct)

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 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 with only one call to **replace**.

The return is a new 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 is a table.

abs (s)

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

s1 in s2

This binary operator checks whether the string `s2` includes `s1` and returns its position as a number.

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). All other characters are left unchanged.

size (s)

with a string s returns its length, i.e. the number of characters in s.

toNumber (e [, base])

Tries to convert its argument to a number. If the argument is already a number or a string convertible to a number, then **toNumber** returns this number; 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 (see 2.1). 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 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 toString calls the corresponding value with e as argument, and uses the result of the call as its result. In Lua, the original name of this function is tostring.

trim (s)

returns a new string with all leading, trailing and excess embedded white spaces removed.

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). All other characters are left 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 Lua, 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`. It also sets a metatable for strings where the `__index` field points to the strings table. Therefore, you can use the string functions in object-oriented style. For instance, strings.repeat(s, i) can be written as s:repeat(i).

`strings.diamap (s)`

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

Example:

```
> strings.diamap('AEIOU-Í_ã+Ï'):
AEIOUÄÖÜÆÅØ
```

If you do not run Agena on Solaris, Windows or Linux, the function simply returns s without any modification.

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

`strings.dump (function)`

Returns a string containing a binary representation of the given function, so that a later **loadstring** on this string returns a copy of the function. `function` must be a Lua function without upvalues.

`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**. 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, 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.

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 Lua interpreter: the string is written between double quotes, and all double quotes, newlines, embedded zeros, and backslashes in the string are correctly escaped when written. For instance, the call

```
strings.format('%q', 'a string with "quotes" and \n new line')
```

will produce the string:

```
"a string with \"quotes\" and \
new line"
```

The options c, d, E, e, f, g, G, i, o, u, X, and x all expect a number as argument, whereas q and s expect a string.

This function does not accept string values containing embedded zeros.

strings.gmatch (s, pattern)

Returns an iterator function that, each time it is called, returns the next captures from `pattern` over string s.

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:

```
table t;
s := 'from=world, to=Lua'
for k, v in strings.gmatch(s, '(%w+)=(%w+)') do
   t[k] := v
od
```

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'
```

`strings.hits (s, pattern)`

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

`strings.isAlpha (s)`

checks whether the string `s` consists entirely of alphabetic letters and return **true** or **false**.

strings.isAlphaNumeric (s)

checks whether the string s consists entirely of numbers or alphabetic letters and return **true** or **false**.

string.isAlphaSpace (s)

checks whether the string s consists entirely of alphabetic letters and/or a space and return **true** or **false**.

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**.

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**.

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**.

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.isMagic2 (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, the diacritics listed at the top of this chapter, and @.

strings.isNumber(s)

checks whether the string s consists entirely of the numbers 0 to 9 and returns **true** or **false**.

strings.isNumberSpace (s)

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

strings.ltrim (s)

returns a new string with all leading white spaces removed.

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.

strings.repeat (s, n)

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

Note that in Lua, the original function name was string.rep.

strings.reverse (s)

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

strings.rtrim (s)

returns a new string with all trailing white spaces removed.

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

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**. 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. If you have to search a string from its beginning, use the faster **in** operator.

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. The function was named string.char in Lua and was renamed to avoid keyword collision with the **char** operator.

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.
- %x: (where x is any non-alphanumeric character) represents the character x. 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 %x 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);
- %b*xy*, where x and y are two distinct characters; such item matches strings that start with x, end with y, and where the x and y are balanced. This means that, if one reads the string from left to right, counting +1 for an x and -1 for a y, the ending y is the first y where the count reaches 0. For instance, the item %b() matches expressions with balanced parentheses.

Pattern:

A *pattern* is a sequence of pattern items. *A* '^' at the beginning of a pattern anchors the match at the beginning of the subject string. A '$' at the end of a pattern anchors the match at the end of the subject string. At other positions, '^' and '$' have no special meaning and represent themselves.

Captures:

A pattern may contain sub-patterns enclosed in parentheses; they describe captures. When a match succeeds, the substrings of the subject string that match captures are stored (captured) for future use. Captures are numbered according to their left parentheses. For instance, in the pattern '(a*(.)%w(%s*))', the part of the string matching 'a*(.)%w(%s*)' is stored as the first capture (and therefore has number 1); the character matching '.' is captured with number 2, and the part matching '%s*' has number 3.

As a special case, the empty capture () captures the current string position (a number). For instance, if we apply the pattern '()aa()' on the string 'flaaap', there will be two captures: 3 and 5.

A pattern cannot contain embedded zeros. Use %z instead.

7.5 Table Manipulation

7.5.1 Kernel Operators

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

`add (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.

`copy (table)`

The operator copies the entire contents of a table into a new table. If the table contains tables itself, those tables are also copied properly (by a `deep copying` method). Metatables are copied, too.

`filled (table)`

checks whether table contains at least one element. The return is **true** or **false**. The function works on dictionaries, as well.

`join (table)`

concatenated all string values in the table in sequential order and returns a string.

`map (f, table [, ...])`

maps the function f on all elements of a table. See map in chapter 7.1 for more information.

`qadd (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.

`unique (table)`

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

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

Please note that the operators returning a Boolean work in a Cantor way, i.e. `{1, 1} = {1}` → true, `{1, 2} xsubset {1, 1, 2, 2, 3, 3}` → true.

`table1 = table2`

This equality check of two tables `table1`, `table2` first tests whether `table1` and `table2` point to the same table reference in memory. If so, it returns **true** and quits.

If not, the operator then checks whether `table1` and `table2` contain the same values without regard to their keys, and returns **true** or **false**. In this case, the search is quadratic.

`table1 <> table2`

This inequality check of two tables `table1`, `table2` first tests whether `table1` and `table2` do not point to the same table reference in memory. If so, it returns **true** and quits.

If not, the operator then checks whether `table1` and `table2` do not contain the same values, and returns **true** or **false**. In this case, the search is quadratic.

`c in table`

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

`table1 intersect table2`

searches all values in `table1` that are also values in `table2` and returns them in a table. The search is quadratic, so you may use **tables.bintersect** instead if you want to compare large tables since **bintersect** performs a binary search.

`table1 minus table2`

searches all values in `table1` that are not values in `table2` and returns them as a table. The search is quadratic, so you may use **tables.bminus** instead if you want to compare large tables since **bminus** performs a binary search.

`table1 subset table2`

checks whether all values in `table1` are included in `table2` and returns **true** or **false**. The operator also returns **true** if `table1` = `table2`. The search is quadratic.

`table1 union table2`

concatenates two tables `table1` and `table2` simply by copying all its elements - even if they occur multiple times - to a new table.

table1 <u>xsubset</u> table2

checks whether all values in `table1` are included in `table2` and whether `table2` contains at least one further element, so that the result is always **false** if `table1 = table2`. The search is quadratic.

7.5.2 tables Library

This library provides generic functions for table manipulation. It provides all its functions inside the table `tables`.

Most functions in the table library assume that the table represents an array or a list. For these functions, when we talk about the 'length' of a table we mean the result of the length operator.

tables.bintersect (table1, table2 [, option])

returns all values of `table1` that are also values in `table2`. The functions performs a binary search in `table2` for each value in `table1`. If no option is given, `table2` is sorted before starting the search. If you pass an option of any value then `table2` should already have been sorted, for no correct results would be returned otherwise.

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

tables.bisEqual (s1, s2 [, option])

Determines whether the table or sequence `s1` contains the same values as the sequence or table `s2`. The functions performs a binary search in `s2` for each value in `s1`. If no option is given (any value), `s2` is sorted before starting the search. If you pass an option of any type then `s2` should already have been sorted, for no correct results would be returned otherwise.

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

tables.bminus (table1, table2 [, option])

returns all values of `table1` that are not values in `table2`. The functions performs a binary search in `table2` for each value in `table1`. If no option is given, `table2` is sorted before starting the search. If you pass the option then `table2` should already have been sorted, for no correct results would be returned otherwise.

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

tables.duplicates (table, option)

returns all the values that are stored more than once to the given table, and returns them in a table. Each duplicate is returned only once. If option is not given, the table is sorted before evaluation since this is needed to evaluate all duplicates. The original table is left untouched, however. If an option of any type is given, the function assumes that the table has been already sorted.

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

`tables.indices (tbl)`

returns all keys in a table as a table. See also: **tables.getvalues** .

`tables.maxn (table)`

Returns the largest positive numerical index of the given table, or zero if the table has no positive numerical indices. (To do its job this function does a linear traversal of the whole table.)

`tables.put (table, [pos,] value)`

Inserts element value at position pos in table, 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, so that a call tables.put(t,x) inserts x at the end of table t.

Use the **insert** *element* **into** *table* statement if you want to add an element at the current end of a table.

`tables.remove (table [, pos])`

Removes from table the element at position pos, shifting down other elements to close the space, if necessary. Returns the value of the removed element. The default value for pos is n, where n is the length of the table, so that a call tables.remove(t) removes the last element of table t.

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

`tables.writeTable (table, filename [, delim])`

writes all values of table `table` to a text file with the given file name with one value in each line. If you pass a delimiter (a string) as the third argument, then the function writes all key~value pairs where the keys and values are separated by the given delimiter.

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

7.6 Set Manipulation

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

`copy (set)`

The operator copies the entire contents of a set 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 (set)`

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

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

Please note that the operators returning a Boolean work in a Cantor way, i.e. `{1, 1} = {1}` → true, `{1, 2} xsubset {1, 1, 2, 2, 3, 3}` → true.

`set1 = set2`

This equality check of two sets `set1`, `set2` first tests whether `set1` and `set2` point to the same set reference in memory. If so, it returns **true** and quits.

If not, the operator then checks whether `set1` and `set2` contain the same items, and returns **true** or **false**. In this case, the search is linear.

`table1 <> table2`

This inequality check of two tables `set1`, `set2` first tests whether `set1` and `set2` do not point to the same set reference in memory. If so, it returns **true** and quits.

If not, the operator then checks whether `set1` and `set2` do not contain the same items, and returns **true** or **false**. In this case, the search is linear.

`c in set`

checks whether `set` contains the item `c` and returns **true** or **false**. The search is constant.

`set1 intersect set2`

searches all items in `set1` that are also items in `set2` and returns them in a set. The search is linear.

`set1` <u>`minus`**</u> `set2`**

searches all items in `set1` that are not items in `set2` and returns them as a set. The search is linear.

`set1` <u>`subset`**</u> `set2`**

checks whether all items in `set1` are included in `set2` and returns **true** or **false**. The operator also returns **true** if `set1` = `set2`. The search is linear.

`set1` <u>`union`**</u> `set2`**

concatenates two sets `set1` and `set2` simply by copying all its items to a new set.

`set1` <u>`xsubset`**</u> `set2`**

checks whether all items in `set1` are included in `set2` and whether `set2` contains at least one further item, so that the result is always **false** if `set1` = `set2`. The search is linear.

7.7 Sequence Manipulation

With the exception of **map**, the following functions have been built into the kernel as unary operators.

`add (seq)`

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

`copy (seq)`

The operator copies the entire contents of a sequence into a new table. If the sequence contains sequence itself, those sequence are also copied properly (by a `deep copying` method). Metatables are copied, too.

`filled (seq)`

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

`join (seq)`

concatenated all string values in the sequence in sequential order and returns a string.

`qadd (seq)`

raises all numeric values in sequence seq to the power of 2 and sums up these powers. The return is a number. If seq 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.

`unique (seq)`

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

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

Please note that the operators returning a Boolean work in a Cantor way, i.e. `seq(1, 1) = seq(1)` → true, `seq(1, 2) xsubset seq(1, 1, 2, 2, 3, 3)` → true.

`seq1 = seq2`

This equality check of two sequences `seq1`, `seq2` first tests whether `seq1` and `seq2` point to the same sequence reference in memory. If so, it returns **true** and quits.

If not, the operator then checks whether `seq1` and `seq2` contain the same values without regard to their keys, and returns **true** or **false**. In this case, the search is quadratic.

`seq1 <> seq2`

This inequality check of two sequences `seq1`, `seq2` first tests whether `seq1` and `seq2` do not point to the same sequence reference in memory. If so, it returns **true** and quits.

If not, the operator then checks whether `seq1` and `seq2` do not contain the same values, and returns **true** or **false**. In this case, the search is quadratic.

`c in seq`

checks whether `seq` contains the value `c` and returns **true** or **false**. The search is linear.

`seq1 intersect seq2`

searches all values in `seq1` that are also values in `seq2` and returns them in a sequence. The search is quadratic.

`seq1 minus seq2`

searches all values in `seq1` that are not values in `seq2` and returns them as a sequence. The search is quadratic.

`seq1 subset seq2`

checks whether all values in `seq1` are included in `seq2` and returns **true** or **false**. The operator also returns **true** if `seq1 = seq2`. The search is quadratic.

`seq1 union seq2`

concatenates two sequences `seq1` and `seq2` simply by copying all its elements - even if they occur multiple times - to a new sequence.

`seq1` <u>`xsubset`</u> `seq2`

checks whether all values in `seq1` are included in `seq2` and whether `seq2` contains at least one further element, so that the result is always **false** if `seq1` = `seq2`. The search is quadratic.

7.8 Mathematical Functions

7.8.1 Kernel Operators

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

`abs (x)`

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

`arctan (x)`

Arc tangent (x in radians). Complex numbers are supported.

`cos (x)`

Cosine (x in radians). Complex numbers are supported.

`entier (x)`

Rounds x downwards to the nearest integer.

`even (x)`

Checks whether x is even. Returns **true** if x is even, and **false** otherwise.

`exp (x)`

returns the value e^x. Complex numbers are supported.

`finite (x)`

Checks whether x is not plus or minus **infinity**, and is not NaN. Returns **true** if x is a "number' and **false** otherwise.

`gammaln (x)`

computes ln Γ x. If x is nonpositive, the function returns **undefined**.

`int (x)`

rounds x to the nearest integer towards zero.

`ln (x)`

Natural logarithm of x. If x is nonpositive, the function returns **undefined**. Complex numbers are supported.

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 real(x) > 0 or real(x) = 0 and imag(x) > 0
- -1, if real(x) < 0 or real(x) = 0 and imag(x) < 0
- 0 otherwise.

sin (x)

Sine (x in radians). Complex numbers are supported.

sqrt (x)

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)

Tangent (x in radians). Complex numbers are supported.

7.8.2 math Library

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

math.arccos (x)

Returns the arc cosine of `x` (in radians). The function works on both numbers and complex values.

math.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.

math.arccoth (x)

Returns the inverse hyperbolic cotangent of `x` (in radians). The function works on both numbers and complex values.

math.arcsin (x)

Returns the arc sine of x (in radians). The function works on both numbers and complex values.

math.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.

math.arctanh (x)

Returns the inverse hyperbolic tangent 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.

math.arctan2 (x, y)

Returns the arc tangent of x/y (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. Complex values are nor supported.

math.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.

math.binomial (n, k)

Returns the binomial coefficient as a number. The function returns **undefined**, if n or k are negative.

math.ceil (x)

Rounds upwards to the nearest integer larger than or equal to 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.

math.conj (z)

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

math.cosh (x)

Returns the hyperbolic cosine of x. The function works on both numbers and complex values.

math.cot (x)

Returns the cotangent -tan(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 numers and complex values.

math.coth (x)

Returns the hyperbolic cotangent 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.

math.csc(x)

Returns the cosecant 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.

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

Differentatiates a function in one variable at the point `x` and returns a number. If eps is not passed, the function uses an accuracy of the value stored to `_EnvEps`. You may pass another numeric value for `eps` if necessary.

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

math.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**. Note that the Agena expression `exp(gammaln(n+1))` is around 25 % faster than this function.

math.flip (z)

swaps the real and imaginary parts of the complex value z = x + I*y and returns the new complex number z' = y + x*I.

math.fmod (x, y)

Returns the remainder of the division of `x` by `y`.

math.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).

math.gcd (a, b)

Returns the greatest common divisor of to numbers a and b as a number. The function is implemented in the Agena language and included in the `library.agn` file.

math.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 accurarcy of `eps` = **_EnvEps** has been reached. You may pass another numeric value for `eps` if necessary.

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

math.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.

math.hypot (x)

returns sqrt(x*x + y*y) 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.

math.irem (x, y)

Evaluates the remainder of an integer division x/y (two Agena numbers). Return is a number. The remainder has the same sign as the numerator.

math.isfloat (x)

Returns **true**, if the number x is a float, i.e. not an integral value, and **false** otherwise.

math.isinteger (x)

Returns **true**, if the number x is an integer, or **false** if it is not.

math.isprime (x)

Returns **true**, if the integral number x is a prime number, and **false** otherwise.

math.lcm(a, b)

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

math.ldexp (m, e)

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

math.log (x, b)

Returns the logarithm of `x` to the base `b`. The function is implemented in the Agena language and included in the `library.agn` file.

math.log10 (x)

Returns the base-10 logarithm of `x`. 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.modf (x)

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

math.Phi

The golden number, Phi = (1+sqrt(5))/2.

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

This function is an interface to the simple pseudo-random generator function `rand` provided by ANSI C. (No guarantees can be given for its statistical properties.)

When called without arguments, returns a pseudo-random real number in the range [0,1). 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)

Sets x as the "seed" for the pseudo-random generator: equal seeds produce equal sequences of numbers.

math.root (x, n)

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

math.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;
```

math.sec(x)

Returns the secant 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.

math.sinh (x)

Returns the hyperbolic sine of x. The function works on both numbers and complex values.

math.sum (f, a, b)

Returns the sum of f(a) + f(a+1) + ... + f(b), where f is a function, as a number.

math.tanh (x)

Returns the hyperbolic tangent of x. The function works on both numbers and complex values.

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. The first one uses implicit file descriptors; 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. The second style uses explicit file descriptors.

The table `io` provides three predefined file descriptors with their usual meanings from C: `io.stdin`, `io.stdout`, and `io.stderr`.

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.close ([file])`

Closes `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 file, closes the default output file.

`io.flush (file)`

`io.flush ()`

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

`io.getkey ()`

reads a key from the keyboard and return its ASCII number. The function works on UNIX and Windows based platforms only. The function is not available on other platforms.

`io.input ([file])`

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 (obj)`

Checks whether `obj` is a valid file handle. Returns true if obj is an open file handle, or **false** if obj is not a file handle.

io.lines ([filename])

io.lines ([file])

In the first form, the function opens the given file name 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`. When the iterator function detects the end of file, it returns **null** (to finish the loop) and automatically closes the file if a filename is given. In case of a file handle, the file is not closed.

The call `io.lines()` (without a file name) is equivalent to io.input()::lines(); that is, it iterates over the lines of the default input file. In this case it does not close the file when the loop ends.

io.open (filename [, mode])

This function opens a file, in the mode specified in the string `mode`. It returns a new file handle, or, in case of errors, **null** plus an error message.

The `mode` string can be any of the following:

- 'r': read mode (the default);
- 'w': write mode;
- 'a': append mode;
- 'r+': update mode, all previous data is preserved;
- 'w+': update mode, all previous data is erased;
- 'a+': append update mode, 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.

io.output ([file])

Similar to **io.input** but operates over the default output file.

io.popen ([prog [, mode]])

Starts program `prog` in a separated process and returns a file handle that you can use to read data from this program (if mode is 'r', the default) or to write data to this program (if mode is 'w').

This function is system dependent and is not available on all platforms.

io.read(file)

io.read ()

In the first form, reads the file `file`, 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 with 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])

reads the entire file with name `filename` 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 on Windows system, diacritics in the file are properly converted to the NT console character set.

Make sure that the lines in the file have no more than 2048 characters, otherwise lines are not correctly split.

If the global system variable _EnvVerbose is set to a value other than null, an error message is printed at the console if the file could not be found.

io.seek (file, [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 file::seek() returns the current file position, without changing it; the call file::seek('set') sets the position to the beginning of the file (and returns 0); and the call file::seek('end') sets the position to the end of the file, and returns its size.

io.setvbuf (file, 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, sizes 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 program ends.

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, all arguments must be strings or numbers. To write other values, use toString or strings.format before write. 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>.

7.10 binio - Binary File Package

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

In this chapter, `filehandle` as the file ID (or file handle) always is a positive integer greater than 2.

binio.close (filehandle [, filehandle2, ...])

closes the files identified by the given file handle(s) and returns **true** if successful, and **fail** otherwise. **fail** will be returned if at least one file could not be closed. 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, **fail** is returned.

binio.length (filehandle)

The function returns the size of the file denoted by `filehandle` in bytes. In case of an error, **fail** is returned.

binio.make (filename)

Creates a file with the given `filename` (a string) in read/write mode and returns a file handle (a number) for subsequent read or write operations. Note that the file is left open. In case of errors, **fail** is returned.

binio.open (filename [, anything])

Opens the given file denoted by `filename` and returns a file handle (a number). If it cannot find the file, the function returns **fail**. The file is 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.

The function also enters the newly opened file into the binio.openfiles table.

See also: **binio.close**.

binio.readchar (filehandle)
binio.readchar (filehandle, position)

In the first form, the function reads a byte from the file denoted by `filename` 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 the **binio.read** function.

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 return is **fail**.

`binio.readnumber (filehandle)`

The function reads an Agena number from the file denoted by `filename` from the current file position and returns it. If there is an error or nothing to read, **fail** is returned.

`binio.readstring (filehandle)`

The function reads a string from the file denoted by `filename` from the current file position and returns it. If there is an error or nothing to read, **fail** is returned.

`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 **fail** otherwise.

See also: **binio.toend**.

`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 **fail** otherwise.

`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 data. The function returns the file position as a number in case of success, and **fail** otherwise.

See also: **binio.rewind**.

`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 **fail** otherwise.

The `number` should be an integer with 0 <= `number` < 256, otherwise `number` % 256 will be stored to the file.

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 **_EnvMinLong** < `number` < _EnvMaxLong, otherwise the operations is not defined.

The function returns **true** in case of success and **fail** otherwise.

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 **fail** 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.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 **fail** 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.beep (freq, dur)`

sounds the loudspeaker with frequency freq (a positive integer) for dur seconds (a positive float). Returns **null** if a sound could be created successfully, or **fail** if wrong arguments were passed.

This function works in Windows only. On UNIX system, there may be a short `beep`.

`os.computername ()`

(Windows only.) Returns the name of the Windows computer. The return is a string. On other architectures, the function returns **fail**.

`os.cd (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.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 Coordinated 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--61), 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.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 nonzero if a shell is available and zero otherwise.

os.exit ([code])

Calls the C function `exit`, with an optional code, to terminate the host program. The default value for code is the success code.

os.fexists (filename)

checks whether the given file (filename is of type string) exists. It returns **true** or **false**.

os.freemem ([unit])

Returns the amount of free physical RAM available on Windows and UNIX machines.

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**.

os.fstat (fn)

Returns information on the file, link (UNIX only), or directory given by the string `fn` in a table of the form `[filetype, size in bytes, [last modification date in the form yyyy, mm, dd, hh, mm, ss]]`. filetype may be `'FILE'` if `fn` is a regular file, `'LINK'` if `fn` is a symbolic link, `'DIR'` if `fn` is a directory, `'CHARSPECFILE'` if `fn` is a character special file (a device like a terminal), `'BLOCKSPECFILE'` if `fn` is a block special file (a device like a disk), or `'OTHER'` otherwise.

os.getenv (varname)

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

os.login ()

(Windows and UNIX only.) Returns the login name of the current user as a string. The return is a string. On other architectures, the function returns **fail**.

os.ls (d [, options])

lists the contents of a directory as a table. If `d` is void, the current working directory is evaluated.

If no option is given, files, links, and directories are returned. If the optional argument `'files'` is given, only files are returned. If the optional argument `'dirs'` is given, only directories are returned. If the optional argument `'links'` is given, only links are returned (UNIX only).

os.lscore (d)

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.

os.md (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.

os.memstate ([unit])

(Windows only.) Returns a table with information on current memory usage on Windows platforms. 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 table contains the keys:

Key	Description
`'freephysical'`	free physical RAM
`'totalphysical'`	installed physical RAM
`'freevirtual'`	free virtual memory
`'totalvirtual'`	total virtual memory

On other architectures, the function returns **fail**.

os.pwd ()

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

os.rd (str)

Deletes 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.

os.rename (oldname, newname)

Renames 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.rm (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.setlocale (locale [, category])

Sets the current locale of the program. `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 Agena is running.

Under Windows, it returns a table containing the string 'Windows', the major version (e.g. 'NT', '2000', etc.) as a string, the Build Number as a number, and the platform ID as a number, in this order.

In UNIX, 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.

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.

7.12 The Debug Library

This library provides the functionality of the debug interface to Lua programs. 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 Lua code (e.g., that variables local to a function cannot be accessed from outside or that userdata metatables cannot be changed by Lua 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.doubleendiantest (n)`

converts a number n (i.e. a C double) twice and returns the converted number, the orginal number, and the difference between the original and the converted values, in this order.

The functions checks the internal function *DoubleToBigEndian* in the C source file `chelpers.c` used by the **binio** package on Little Endian platforms to write and read Agena numbers to/from file. If you should encounter trouble with Agena compiled with GCC on Little Endian hardware, then you might try the `-DGCC_WROUNDOFF_BUG` compilation option. The switch assumes, that on your platform, doubles consist of eight bytes.

`debug.getfenv (o)`

Returns the environment of object `o`.

`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.

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 (func, up)

This function returns the name and the value of the upvalue with index `up` of the function func. The function returns **null** if there is no upvalue with the given index.

debug.setfenv (object, table)

Sets the environment of the given `object` `to the given` `table`. 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 Lua calls a function;
- 'r': The hook is called every time Lua returns from a function;
- 'l': The hook is called every time Lua 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, Lua 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, table)

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

debug.setupvalue (func, up, value)

This function assigns the value `value` to the upvalue with index up of the function func. 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, float, double), 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 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.

`utils.wait (x)`

waits for x seconds. 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 UNIX and Windows based systems only. On other architectures, the function returns **fail**.

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 than the function assumes that all values in `t` are sorted in ascending order so that execution is much faster.

`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 compress - Text Compression

This package contains functionality for compressing and decompressing texts. *ACF* means 'Agena Compressed File'. As a *plus* package, it is not part of the standard distribution and must be activated with the **readlib** or **with** functions.

Please note that this package is still experimental.

`compress.attrib (filehandle)`

returns information on an ACF denoted by its numeric `filehandle` as a table with its keys denoting the following information as their values:

Key	Description	Type
'commentpos'	The position of a comment in the base. Its value is always 0.	number
'creation'	The date of creation of the base. The return is a formatted string including date and time.	string
'description'	The description of the ACF file (a string). If no description is given, a string with 75 spaces is returned.	string
'method'	The method used: 'huffman' or 'lz77'.	string
'size_compressed'	The number of bytes of the compressed data.	number
'size_uncompressed'	The number of bytes of the original uncompressed data.	number
'stamp'	The stamp at the beginning of the file.	string
'version'	The ACF version.	number

`compress.create (filename, str [, method [, description]])`

creates an ACF with the given `filename`. The function takes the uncompressed data in string `str`, compresses it and writes the compressed data along with some administrative information to the file.

The following compression methods are available:

- Huffman compression (the default). You may also pass the string `'huffman'` as a third argument.

- LZ77 compression developed by Abraham Lempel and Jakob Ziv in 1977. Pass `'lz77'` explicitly as a third argument to the function to use this method.

A description with a text of up to 75 characters can also be included in the ACF. Pass the text as a string as the optional fourth argument. The method must be given if you want to add a description to the file.

If the data could be compressed and written successfully to the file, **true** is returned. Otherwise **fail** will indicate an error which may occur generally if not enough memory has been available.

`compress.close (filehandle [, filehandle2, ...])`

closes one or more ACFs given by their file handles. If all files could be closed successfully, **true** is returned, else **false**. The function also deletes the files successfully closed from the **compress.openfiles** table.

See also: **compress.close** .

`compress.desc (filehandle)`
`compress.desc (filehandle, description)`

In the first form, returns the description of an ACF stored in the file header.

In the second form, **compress.desc** sets or overwrites the description section of an ACF. 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**.

See also: **compress.attrib**.

`compress.inflate (str, method)`

compresses the data in the string `str` with the given `method` (a number). The method can be either 0 (zero) for Huffman compression or 1 for LZ77 compression.

This function - although correctly working - is for experimental use only and may be changed or deprecated in the future.

`compress.lines (filehandle)`

decompresses the data in the ACF file denoted by its `filehandle` and returns an iterator function that every time it is called, returns a new line with a decompress string.

If no more lines can be returned (EOF has been reached), the function will always return **null**.

See also: **compress.read**.

`compress.open (filename)`

opens an ACF given by its `filename` and returns the file handle as a number. If the file cannot be found, **fail** is returned. The function also adds the name of the opened ACF to the **compress.openfiles** table.

The method with which the data in the file has been compressed is automatically detected. The file is opened in read-only mode, since no data ca be added or deleted to a once created file.

See also: **compress.close** .

`compress.openfiles`

A global table containing all ACFs currently open. Its keys are the file handles (integers), the values the file names (strings). If there are no open files, **compress.openfiles** is an empty table.

`compress.ratio (filehandle)`

returns the compression rate as a number. The compression ratio is the compressed size divided by the uncompressed size of the data in the file denoted by its `filehandle`.

See also: **compress.attrib** .

`compress.read (filehandle)`

reads and decompresses all the data in the ACF denoted by its `filehandle`. It returns the data as a string if everything went right, and **fail** otherwise.

See also: **compress.lines** .

`compress.stringtoset (str)`

takes a string and returns all lines in the string (denoted by the newline character '\n') as separate items in a set.

7.16 calc - Calculus Package

This package contains mathematical routines to perform basic calculus. As a *plus* package, it is not part of the standard distribution and must be activated with the **readlib** or **with** functions.

`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 `_EnvEps`. You may pass another numeric value for `eps` if necessary.

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

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

creates a sequence **seq(**1~`f`(a), 2~`f`(a+1), ..., (`b`-`a`+1)~`f`(`b`)**)**, with `f` a function, `a` and `b` numbers. Thus, the function `f` is applied to all numbers between and including `a` and `b`. The step size is 1.

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

computes the sum of `f`(a), ..., `f`(`b`), with `f` a function, `a` and `b` numbers. If `a` > `b`, then the result is 0.

`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 accurarcy of `eps` = **_EnvEps** has been reached. You may pass another numeric value for `eps` if necessary.

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

`calc.interp (tp)`

computes a Newton interpolating polynomial as a function. The interpolation points are passed in a table tp, with each point being represented as a pair x_k:y_k.

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

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

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

The function divides the interval [`a`, `b`] into smaller intervals [`a`, `a`+`step`], [`a`+`step`, `a`+2*`step`], ..., [`a`+p*`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`=_EnvEps as a default. `f` must be differentiable on [a, b].

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

7.17 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 constructed with the above mentioned constructors.

The package includes a metatable linalg.vmt 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.

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.

`abs (A)`

determines the length of vector A. This operation is done by applying a metamethod to A.

`linalg.backsubs (A, B)`

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

`linalg.coldim (A [, ...])`

determines the column dimension of the matrix A. The return is a number.

If you pass more than one argument, then the 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 are not checked, if you pass more than one matrix. See **linalg.ismatrix** for information on how this check is being done.

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 went fine, the function returns the dimensions of all vectors passed.

See **linalg.isvector** for information on how this 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 the 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 (A)

computes the cross-product of two vectors 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 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 the number of rows and the right-hand side the number of columns. If A is a vector, the size of the vector is determined.

`linalg.dotprod (v1, v2)`

computes the vector dot product of two vectors v1, v2 of same dimension. The vectors must consist of Agena numbers. The return is a number.

`linalg.hilbert (n [, x])`

creates a generalized n x 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.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.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.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 (o1, o2, ..., on)`

creates a matrix of the structures o[k] given. Valid structures are vectors created with **linalg.vector**, tables, or sequences.

The return is a table with the user-defined type 'matrix' and a metamethod defined by **linalg.mmt**.

`linalg.mmap (f, A [, ...])`

This function maps a function f or anonymous function to all the components in the matrix A and returns a new matrix. The function must return only one value. See **linalg.vmap** for further information.

`linalg.mmul (A, B)`

conducts a multiplication of a m x n- and a n x p-matrix and returns a m x p matrix.

`linalg.rowdim (A [, ...])`

determines the row dimension of the matrix A. The return is a number.

If you pass more than one argument, then the time-consuming check whether A is a matrix, is skipped.

A more direct way of determining the column dimension is `left(A.dim)`.

See also: linalg.coldim .

`linalg.scalarmul (A, n)`

performs a scalar multiplication by multiplying each element in vector A with the number n. The result is a vector.

`linalg.transpose (A)`

computes the transpose of a m x n-matrix A and thus returns an n x m-matrix.

`linalg.vadd (A, B)`

determines the vector sum of vector A and vector B. The return is a vector.

See also: linalg.vsub .

`linalg.vector (a1, a2, ...)`

`linalg.vector ([a1, a2, ...])`

`linalg.vector (seq(a1, a2, ...))`

`linalg.vector (n, [a1, a2, ...])`

`linalg.vector (n, [ ])`

creates a vector with numeric components a1, a2, etc. The function also accepts a table or sequence of elements a1, a2, etc. (second and third form).

In the fourth form, n denotes the dimension of the vector. 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 with the user-defined type 'vector' and a 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.vsub (A , B)

subtracts vector B from vector A. The result is a vector.

See also: **linalg.vadd**.

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 s[k] := f(v1[k], v2[k]).

v1 and v2 must have the same dimension.

See also: **linalg.vmap**.

linalg.zero (n)

creates a zero vector of length n with all ist components physically set to 0. If you want to create a sparse zero vector of dimension n, enter: `linalg.vector(n, [])`.

Chapter Eight

Agena Database System

8 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 long integer, 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.

Note that by setting the global system variable **_EnvVerbose** to **null**, some non-critical warning messages are suppressed.

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 add.size 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.size**.

`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 shrinked 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**. or If a file truncation error occurred, clean returns **undefined**.

`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, **undefined** is returned.

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, **fail** is returned, and no comment is written. **fail** is also 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, **fail** is returned and no base is created.
number_of_columns	The number of columns in a database. Default: 2 (key and value). If the base is not a database, this option is ignored. If the number of columns is nonpositive, **fail** 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 or not a database or list, and **true** otherwise.

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 or if the base is a sequence, **fail** is returned. If the base is empty, **null** is returned. See also: ads.get, ads.getvalues.

ads.getvalues (filehandle [, column])

by default gets all valid entries in the second column in a database and returns them in a table. If the optional argument column is given, the entries in this column are returned. Argument: file handle (integer). If the file is not open or if the column does not exist, **fail** is returned. If the base is empty, **null** is returned. With lists, the return is always **null**. See also: ads.get, ads.getkeys.

ads.index (filehandle, key)

searches for the given key (a string) in the base pointed to by filehandle and returns its file position as a number. If their are no entries in the set, the function returns **null**. If the file is not open, **fail** is returned.

ads.indices (filehandle)

returns the file positions of all valid detests as a table.

If the file is not open, indices returns **fail**. If there are no entries in the base, the return is an empty table, otherwise a table with the indices is returned. 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.

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.openbase (filename [, anything])

Opens the base with name filename and returns a file handle (a number). If it cannot find the file, or the base has not the correct version number, the function returns **fail**. The base is opened in both read and write mode.

If an optional second argument is given (any valid Agena value), the base is opened in read mode only.

The function also enters the newly opened file into the ads.openfiles table.

ads.openfiles

A global table containing all files currently open. Its keys are the file handles (integers), the values the file names (strings). If there are no open files, ads.openfiles is an empty table.

ads.peek (filehandle, position)

returns both the length of an entry (including the terminating \0 character) and the entry itself at the given file position as two values (an integer and a string). The function is save, so if you try to access an invalid file position, the function will exit returning **fail**.

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, read 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. **fail** is also returned of the base is not a database or a list.

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 file handle, first argument) at the given file position (an integer, second argument). Two values are returned: the respective key and its value. With lists, only the key is returned.

The function is save, so if you try to access an invalid file position, the function will exit returning **fail**.

If the file is not open, retrieve returns **fail**, as well If the database or list is empty, **null** is returned. See also ads.indices, ads.invalids.

`ads.sizeof (filehandle)`

Returns the number of valid records (an integer) in the base pointed to be filehandle. If the base pointed to by the numeric filehandle is not open, the function returns **fail**.

`ads.sync (filehandle)`

flushes all unwritten content to the base file. The function returns **true** if successful, and **fail** otherwise (e.g. if the file was not opened before or an error during flushing occurred).

`ads.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**.

Chapter Nine

C API Functions

9 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 14: Modified Lua C API functions

Except for the above mentioned functions, no other modifications have been made to Lua C API functions that have been shipped with 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 added (see files `lapi.c` and `lua.h`):

agn_ccall

```
agn_Complex agn_ccall (lua_State *L, int nargs, int nresults);
```

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, 0 is returned. **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_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` preallocated places (`nrec` may be zero).

agn_createset

```
void agn_createset (lua_State *L, int nrec);
```

Pushes an empty set onto the top of the stack. The new set has space pre-allocated for `nrec` items.

agn_deletertable

```
LUA_API void agn_deletertable (lua_State *L, int objindex)
```

Deletes the remember table of the procedure at stack index `idx`. If the procedure has no remember table, nothing happens. The function leaves the stack unchanged.

agn_fnext

```
int agn_fnext (lua_State *L, int indextable, indexfunction);
```

Pops a key from the stack, and pushes four values in the following order: the key of a table given by indextable, its corresponding value, the function at stack number

indexfunction, and the value from the table at the given indextable. If there are no more elements in the table, then **agn_fnext** returns 0 (and pushes nothing).
The function is useful to avoid duplicating values on the stack for **lua_call** and the iterator to work correctly.

A typical traversal looks like this:

```
/* table is in the stack at index 't', function is at stack index 'f' */
lua_pushnil(L); /* first key */
while (lua_fnext(L, t, f) != 0) {
  /* '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_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_getseqlstring

```
const char *agn_getseqlstring (lua_State *L, int idx, int n, size_t *l);
```

gets the string at index `n` in the sequence at stack index `idx`. The length of the string is stored to `l`.

agn_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_gettstring

```
const char *agn_gettstring (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 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_isutype

```
int *agn_isutype (lua_State *L, int idx, const char *str);
```

Checks whether the type at stack index `idx` is the user-defined type denoted by `str`. It returns 1 if the given user-defined type has been found, and 0 otherwise.

agn_isutypeset

```
int *agn_isutypeset (lua_State *L, int idx, const char *str);
```

Checks whether a user-defined type has been set for the given object at stack position idx. It returns 1 if a user-defined type has been set, and 0 otherwise. The function accepts any Agena types. By default, if the object is not a sequence, a pair, or procedure, it returns 0.

agn_issequtype

```
int *agn_issequtype (lua_State *L, int idx, const char *str);
```

Checks whether the type at stack index `idx` is a sequence and whether the sequence has the user-defined type denoted by `str`. It returns 1 if the above condition is true, and 0 otherwise.

agn_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_isverbose

```
LUA_API int agn_isverbose (lua_State *L);
```

Checks whether the global system variable **_EnvVerbose** is set to anything but **null** or **false**. If **_EnvVerbose** is set, the function returns 1, otherwise (**_EnvVerbose** is unassigned or **false**) 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, 0 is returned. **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 at t[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 index);
```

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 `index` 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.

agn_pairrawset

```
void agn_pairrawset (lua_State *L, int index);
```

Does the equivalent to p[k] := v, where s is a pair 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.

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

```
int 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_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, 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 or pair also causes the pretty printer to display the string passed to the function instead of the usual output at the console. This does not apply to procedures.

See also: **agn_getutype**.

agn_setutypestring

```
void agn_setutypestring (lua_State *L, int idxobj, const char *str);
```

Sets the string `str` as the user-defined type of the procedure, sequence, or pair at stack position `idxobj`.

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[])
```

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 four 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]. The number of allocable key~value pairs to the array part is stored to a[2], and the number of allocable key~value pairs to the hash part is stored to a[3].

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.

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.

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] = num, where t is the value at the given valid index idx, n is an integer, and num an Agena number (a C double).

This function leaves the stack unchanged. The assignment is raw; that is, it does not invoke metamethods.

lua_rawsetistring

```
void lua_rawsetistring (lua_State *L, int idx, int n, const char *str);
```

Does the equivalent of t[n] = str, where t is the value at the given valid index idx, n is an integer, and str a string.

This function leaves the stack unchanged. The assignment is raw; that is, it does not invoke metamethods.

lua_rawsetstringlint

```
void lua_rawsetstringlint (lua_State *L, int idx, const char *str,
   int len, int n);
```

Does the equivalent of t[str] = n, where t is the value at the given valid index idx, str a string, len the length of str, and n an integer.

This function leaves the stack unchanged. The assignment is raw; that is, it does not invoke metamethods.

lua_rawsetstring number

```
void lua_rawsetstringnumber
   (lua_State *L, int idx, const char *str, lua_Number n);
```

Does the equivalent of t[str] = n, where t is the value at the given valid index idx, `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 table 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.

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_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_sget

```
void lua_sget (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.

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_srawset

```
void lua_srawset (lua_State *L, int index);
```

Does the equivalent to insert v into s, where s is the value at the given valid index index, 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_srawget

```
void lua_srawget (lua_State *L, int index);
```

Pushes onto the stack the set value t[k], where t is the table 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_usnext

```
int lua_usnext (lua_State *L, int index);
```

Pops a key from the stack, and pushes the next item twice (!) from the set at the given index. 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_checkudata (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.

Appendix

A1 Operators

Unary operators are:

`abs`, `add`, `arctan`, `assigned`, `char`, `copy`, `cos`, `entier`, `even`, `exp`, `filled`, `finite`, `gammaln`, `imag`, `int`, `isnull`, `join`, `left`, `ln`, `lower`, `nargs`, `not`, `qadd`, `real`, `replace`, `right`, `sign`, `sin`, `size`, `sqrt`, `tan`, `trim`, `type`, `unique`, `upper`, `utype`, `-` (unary minus).

Binary operators are:

`in`, `intersect`, `minus`, `shift`, `split`, `subset`, `union`, `xsubset`, `+` (addition), `-` (subtraction), `*` (multiplication), `/` (division), `%` (modulus), `^` (exponentiation), `**` (integer exponentiation), `..` (concatenation), `=` (equality), `<` (less than), `<=` (less or equal), `>` (greater than) , `>=` (greater or equal), `$` (substring), `:` (pair constructor), `!` (complex constructor), `\` (integer division).

A2 Metamethods

The following metamethods were inherited from Lua 5.1:

Index to metatable	Meaning
'__index'	Method for keys (for tables and sets only) in read operations in tables or sets.
'__newindex'	Procedure invoked when a value shall to be written to a table or set.
'__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 15: 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'	**abs** operator
'__arctan'	**arctan** operator
'__cos'	**cos** operator
'__entier'	**entier** operator
'__even'	**even** operator
'__exp'	**exp** operator
'__finite'	**finite** operator
'__gammaln'	**gammaln** operator
'__in'	in binary operator (for tables and sequences only)
'__int'	**int** operator
'__intdiv'	integer division
'__ipow'	exponentiation with an integer power
'__ln'	**ln** operator
'__qsadd'	**qadd** operator for table or sequence based user-defined types
'__sadd'	**add** operator for table or sequence based user-defined types
'__shift'	**shift** operator
'__sign'	**sign** operator
'__size'	**size** operator
'__sin'	**sin** operator
'__sqrt'	**sqrt** operator
'__tan'	**tan** operator

Table 16: Metamethods introduced with Agena

A3 System Variables

Agena lets you configure the following settings:

System variable	Meaning
_EnvAgenaPath	path to the main Agena directory
_EnvMaxLong	The maximum integral value of the C type long on your platform; do not change this value.
_EnvMinLong	The minimum integral value of the C type long on your platform; do not change this value.
_EnvMore	number of entries in tables and sets printed by **print** and the end-colon functionality before issuing the \`press any key\` prompt.
_EnvPrintNewLineAfterInput	If set to true, a newline is printed at the console after entering a statement. Default: unassigned, i.e. no newline.

System variable	Meaning
_EnvPrintZeroedCmplxVals	When set to true, real and imaginary parts of complex values close to zero are rounded to zero on output. (Note that internally, complex values are not rounded.)
_EnvWithProtected	table of names (passed as strings) that cannot by overwritten by the with function.
_EnvWithVerbose	If set to false, the **with** function will not display warnings, the init string, and the short names assigned.
_PROMPT	Defines the prompt Agena displays at the console
_RELEASE	the Release of Agena returned as a string, e.g. 'Agena 0.12'

Table 17: System variables