

agenda > >

a programming language

---

**primer**

for version 7.5 tethys

by alexander walz

june 03, 2026

agena Copyright 2006 to 2026 by alexander walz, rhineland.  
All rights reserved. Portions Copyright 1994-2025 Lua.org, All rights reserved.

None of the Agena project members or anyone else connected with this documentation, in any way whatsoever, can be responsible for your use of the information contained in or linked from it.

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

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

This manual has been created with Lotus Word Pro 98 running on Sun Microsystems VirtualBox with Microsoft Windows 2000 Professional, Visio 2013, yWorks yEd Graph Editor, and PDF Creator.

## Credits

### The Sources

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

### Chapter 7: Standard Library documentation

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

#### **environ.anames**

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

#### **case of** statement

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

#### **next** statement

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

#### **environ.globals** base library function

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

#### **mkdir**, **chdir**, and **rmdir** functions in the **os** library

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

## No automatic auto-conversion of strings to numbers

was inspired by Thomas Reuben's `no_auto_conversion.patch` available at [lua.org](http://lua.org).

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

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

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

taken from John Hind's Lua 5.1.4 patch available at [lua.org](http://lua.org).

## Integer division

taken from Thierry Grellier's `newluaoperators.patch` available at [lua.org](http://lua.org).

## math.fraction

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

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

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

## calc.diff

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

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

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

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

### **calc.minimum, calc.maximum**

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

### **bernoulli, besselj, bessely, euler, lambda**

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

## **Graphics**

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

## **ADS package**

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

## **MAPM binding**

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

The MAPM Agena binding is an adaptation of the Lua binding written by Luiz Henrique de Figueiredo, put to the public domain. As Mike Ring unfortunately passed away in December 2011, you are welcome to propose a Lua C extension of Henrique's binding.

## Year 2038 fix for 32-bit machines

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

## gzip package

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

## Internal string concatenation

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

**\*\* operator and the functions `arctan`, `expx2`, `fact`, `gamma`, `lngamma`, `calc.Ai`, `calc.bessel0`, `calc.bessel1`, `calc.Bi`, `calc.dawson`, `calc.dilog`, `calc.Ci`, `calc.Chi`, `calc.elliptic1`, `calc.elliptic2`, `calc.En`, `calc.fresnelc`, `calc.fresnels`, `calc.hyp1f1`, `calc.hyp2f1`, `calc.ibeta`, `calc.igamma`, `calc.igammc`, `calc.invibeta`, `calc.jacobian`, `calc.polylog`, `calc.Psi`, `calc.Si`, `calc.Shi`, `calc.Ssi`, `calc.zeta`, `calc.zeta2`, `math.cosd`, `math.cotd`, `math.sind`, `math.tand`, `stats.F`, `stats.Fc`, `stats.invF`, `stats.gammad`, `stats.gammadc`, and `stats.invnormald`**

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

## **`erf`, `erfc`, `inverf`, `inverfc`, `calc.intcc`, `calc.intde`, `calc.intdei`, `calc.intdeo`**

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

## **`math.random`**

The algorithm used to compute random numbers has been written by George Marsaglia and published on [en.wikipedia.org](http://en.wikipedia.org).

## **`io.anykey`**

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

## **xBASE file support**

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

## **The net package**

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

`Beej's Guide to Network Programming, Using Internet Sockets`, written by Brian "Beej Jorgensen" Hall, was of great help, also when extending all the functionality to IPv6. Some of the **net** functions use part of Mr. Hall's public domain code published in his tutorial. Copyright © 2009, 2025 "Beej Jorgensen" Hall.

Studying the code of the LuaSocket 2.0.2 package, Copyright © 2004-2007 by Diego Nehab, and published under the MIT licence, was very worthwhile. Also, **net.address**, **net.getaddrinfo**, **net.remoteaddress**, **net.tohostname** use code taken from Diego Nehab's MIT-licenced LuaSocket 3.1.0 package, Copyright © 2004-2022 Diego Nehab.

## **strings.dleven**

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

## **utils.readxml**

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

## **utils.decodeb64 and utils.encodeb64**

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

## **printf**

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

## **.. operator and {} indexing**

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

## **copy**

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

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

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

## **bags package**

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

## **xml package**

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

LuaExpat development was sponsored by Fábrica Digital and FINEP.

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

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

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

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



## **linalg.scale, stats.scale**

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

**linalg.rotcol, linalg.rotrow, linalg.infnorm, linalg.infcolnorm, linalg.matinfnorm, linalg.matinfnorm, linalg.matmat, linalg.matnnorm, linalg.matonenorm, linalg.mattam, linalg.ncolnorm, linalg.onecolnorm, linalg.nnorm, linalg.onenorm, linalg.scale, linalg.tridecomp, stats.scale**

have all been ported from ALGOL 60 to C by the author, taken from the ALGOL 60 NUMAL package published by The Stichting Centrum Wiskunde & Informatica (Stichting CWI) (legal successor of Stichting Mathematisch Centrum) at Amsterdam. The NUMAL package has been developed and released in the late 1960s and early 1970s.

## **os.now** and the **astro** package

use C routines of the IAU Standards of Fundamental Astronomy (SOFA) Libraries, See Appendix B5 for the licence. The Persian data/time functions have been created by Gemini AI, put to the public domain. The **astro** package also uses functions written by Don Cross, available in his remarkable MIT-licensed library `Astronomy Engine for C/C++`.

Functions **calc.clamped spline, calc.clamped spline coeffs, calc.interp, calc.neville, calc.newton coeffs, calc.nokspline, calc.nokspline coeffs**

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

## **stats.smallest**

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

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

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

**astro.moonriset**

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

**astro.phase**

Uses C functions taken from: [http://www.voidware.com/moon\\_phase.htm](http://www.voidware.com/moon_phase.htm). There have not been any copyright remarks.

**astro.sunriset**

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

**astro.cdate** & **astro.jdate**

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

**strings.utf8size**

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

**strings.isutf8**

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

**strings.isotolatin** & **strings.isotoutf8**

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

## **strings.glob**

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

## **stats.sorted**

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

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

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

## **E, Exp**

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

## **Complex arithmetic**

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

## **io.getclip** and **io.putclip**

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

## **try/catch** statement

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

## **debug.getinfo**

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

**calc.polyfit & calc.linterp**

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

**Review of the Agena interpreter at the Web**

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

Many thanks also to a very kind and very benignly strict contributor from the State of Israel. It helped so much understanding what I was actually doing with this project.

**linalg.adjoint, linalg.permanent, linalg.inverse & linalg.minor**

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

**redo & relaunch**

have been inspired by the Ruby programming language.

**linalg.linsolve**

is based on C functions written by Edward Popko and Alexander Evans; for the former see the link above, and for the latter the following address: <http://www.dailyfreecode.com/code/basic-gauss-elimination-method-gauss-2949.aspx>.

**Infact, dblfact, trifact, calc.Cin, calc.eta, calc.auxSiCi, calc.simaptive and linalg.ludoolittle**

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

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

use methods developed by Donald Knuth.

### **calc.Ei & calc.Ein**

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

### **linalg.ref**

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

### **linalg.forsub**

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

### **cordic package**

is based on a C package written by John Burkardt, taken from [http://people.sc.fsu.edu/~jburkardt/c\\_src/cordic/cordic.c](http://people.sc.fsu.edu/~jburkardt/c_src/cordic/cordic.c), with modifications done using Maple V Release 4 and a TI-Nspire CX CAS. MIT-licenced.

### **libusb binding**

is based on `lua-libusb1` - Lua binding for libusb 1.0, written by Tom N Harris. See: <http://lua-libusb1.googlecode.com>.

### **stats.extrema**

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

### **mdf, xdf**

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

### **os.cputload, os.drivestat, os.getenv, os.realpath & os.setenv**

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

### **utils.readini & ini package**

use C functions written by Nicolas Devillard for his iniparser package, May 2024 edition, MIT licenced.

### **Various OS/2 operating system functions**

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

### **llist & heaps packages**

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

### **stats.dbscan & stats.neighbours**

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

### **hashes package**

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

### **math.ceilpow2 and math.ilog10**

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

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

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

## **os.terminate**

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

## **os.monitor**

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

## **hypot2 and antilog<sub>n</sub> operators**

have been inspired by the Sinclair Scientific Programmable pocket calculator.

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

have been inspired by Matlab.

## **stats.gmean**

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

## **gdi.plotfn**

has been improved by Slobodan from Serbia.

## **oftype metamethod**

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

## **stats.durbinwatson, stats.standardise, and stats.sumdataIn**

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

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

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

## Chapter 6.24

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

### Exit and restart handling

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

### with and related statements

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

### math.dms

uses an algorithms proposed by user807566 on StackOverflow.

### case of *boolean condition variant*

has been inspired by the Go programming language.

### Numeric ranges in case/of clauses

have been inspired by the Fortran 90 programming language.

### math.fma

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

### math.signbit

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

### math.signbit

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



### **numtheory.kronecker**

has been written by Harry J. Smith, published on [alt.math.recreational](http://alt.math.recreational) in 2004.

### **math.wrap**

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

### **Sinclair ZX Spectrum package**

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

### **math.eps**

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

### **dBASE version numbers**

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

### **round, mdf, and xdf**

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

### **math.cld, math.fld, math.flipsign, math.isqrt, math.lnfact, and numtheory.powmod**

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

### **strings.appendmissing, strings.between, strings.chop, strings.chomp, strings.contains, strings.uncapitalise, strings.iswrapped, strings.wrap**

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

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

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

**hashes.mix64**, **hashes.mix64to32**, **hashes.wang**

use Thomas Wang's C procedures, taken from [gist.github.com/badboy/6267743](https://gist.github.com/badboy/6267743).

**times**

is based on the corresponding Haskell function `iterate`.

**for/until** loops

have been inspired by COBOL.

**math.sincos**

uses Elliot Saba's `sincos` implementation.

**math.accu**

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

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

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

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

are based on C functions written by Arash Partow.

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

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

### **bloom** filter plus package

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

### **factory** plus package

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

### **strings.a64**, **hashes.sha256** and **hashes.sha512**

use C code from the musl-1.1.19/1.2.4 libraries, MIT licenced.

### **? statement**, **prepend**, **linalg.iszero**, **linalg.isone**, thus indirectly **satisfy**

have been inspired by the Axiom Computer Algebra System.

### **getorset**

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

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

have been inspired by Algol 68.

### **bytes.pack**, **bytes.packsize**, **bytes.unpack**, **tables.move**, **utf8**

have been taken from Lua 5.3.5 or Lua 5.4.0 RC 4/5.4.8 (**utf8**, **move**).

### **GMP 6.1.2 port for OS/2**

compiled by KO Myung-Hun has been used to compile the **gmp** binding.

### **dual package**

uses definitions primarily found at [blog.demofox.org](http://blog.demofox.org) and [adl.stanford.edu](http://adl.stanford.edu).

**os.iterate**

has been derived from a listing published in `Programming in Lua` 2nd Ed., pp 271f., by Roberto Ierusalimsky.

**com package**

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

**assignments in conditions of while loops, if and case of statements**

were inspired by Icon and C.

**duplicate parser warnings for duplicate local variable declaration**

have originally been designed by Domingo Alvarez Duarte for Lua 5.1.

**shift**

has been written by StackOverflow user ryanpattison for Lua.

**type anything** and more or less **constants**

have been inspired by Maple.

**erfcx, calc.scaled Dawson, calc.w**

use code written by Steven G. Johnson, October 2012, MIT licence.

**os.getip, os.netuse, os.netsend & os.netdomain**

use code written by Antonio Escaño Scuri for the NTLua 3.0 package, MIT licence. Non-Windows code in **os.getip** by Smitha Dinesh Semwal.

**utils.decodeb85, utils.encodeb85, utils.decodez85, utils.encodez85**

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

## **utils.decodea85 and utils.encodea85**

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

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

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

## **bimaps package**

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

## **heaps package**

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

## **factory.curry function**

has originally been written by Rici Lake for Lua 5.x.

## **tuples package**

is based on functions written by Roberto Ierusalimsky.

## **strings.walker**

implements C code written by John Walker, Fourmilab.

## **iconv package**

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

## **factory.anyof & environ.callable**

have both originally been conceived and written by Gary V. Vaughan in Lua,  
included in his lyaml package for Lua 5.x, MIT licenced.

## **regex package**

has originally been written by Reuben Thomas and Shmuel Zeigerman for Lua 5.1 to 5.4, 2000 - 2020, MIT licence. They are also the authors of the documentation.

## **json package**

has originally been written by David Heiko Kolf for Lua 5.1+, Copyright (C) 2010-2021, MIT licenced.

## **AgendaEdit GUI**

The GUI is based on an editor published under the GPL licence and written by Bill Spitzak and others for FLTK 1.3 <http://www.fltk.org>.

Thanks to Albrecht Schlosser for making the editor work with Agenda.

**erf** (2-arg mode), **math.chi**, **stats.binompdf**, **stats.binomd**, **stats.poisson** and **stats.poissond**

have been inspired by Texas Instrument's Derive 6.1 Computer Algebra System.

**calc.Psi** in 2-arg mode uses MIT-licenced C functions written by Tom Minka.

**calc.gammainc**, **stats.gammcdf** and **stats.gammapdf**

are based on code written by CRBond, (C) 1993, C. Bond. All rights reserved.

**combinat.choose** and **combinat.permute**

are ports of functions of the same name as found in Maple V Release 4, Copyright (c) 1991 by the University of Waterloo. All rights reserved.

**The Red-black tree** implementation

has been written by Mathieu Rabine, MIT licenced.

The **Base32** implementation of **utils.decodeb32** and **utils.encodeb32**

has been written by Copyright (c) 2010 Adrien Kunysz, MIT licenced.

**linalg.eigen** & **linalg.eigenval**

have originally been written by Copyright (c) 1996 Frank Uhlig et al. and 2009 Genome Research Ltd (GRL), MIT licenced.

**linalg.det**, **numtheory.gcd** and **numtheory.lcm**

use code presented and well explained at GeeksForGeeks of Noida, Uttar Pradesh, Republic of India.

**Recursive descent algorithm**

for nested tables used by functions **map** and **subs** has been originally written in C by Chaos, Shanghai, PRC, and posted on StackOverflow in 2020

**utils.rfc3339**

is based on C code written by Matteo Benzi, published under the MIT licence.

**Mixing classical indexing and OOP method calls with the ``__index'` tag**

has been inspired by Luther's response in the StackOverflow article ``Specifying both "methods" and index operator in Lua metatable``.

**os.period**, **os.timestamp**, **os.ticker**, **math.noise**, **tables.cleanse**

are all based on code written for Luau, a Lua derivative, MIT licenced, Copyright (c) 2019-2024 Roblox Corporation.

**curses**

is a 1:1 port of the lcurses binding for Lua 5.1 written by Reuben Thomas & Tiago Dionizio under the MIT licence.

## **fzy package**

is an adaption of the Lua package of the same name written by Seth Warn which itself binds to John Hawthorn's fzy C library.

For the original fzy C library: Copyright (c) 2014 John Hawthorn, MIT licence.

For the Lua binding: Copyright (c) 2020 Seth Warn, MIT licence.

## **kiss FFT package**

is a port of Benjamin von Ardenne's LuaFFT package which itself is the Lua port of the KissFFT Library by Mark Borgerding for C. MIT-licenced.

## **Correlation, zero-finding and sine/cosine transform functions**

**stats.besselj**, **stats.besselk**, **stats.brownian**, **stats.circular**, **stats.constant**, **stats.cubic**, **stats.dampedcos**, **stats.dampedsin**, **stats.exponential**, **stats.gaussian**, **stats.hole**, **stats.linear**, **stats.matern**, **stats.penta**, **stats.power**, **stats.ratquad**, **stats.spherical**, **stats.white**, **calc.brent**, **calc.cdf**, **calc.cst**, **calc.chandrupatla** and **calc.itp** are all based on MIT-licenced code written by John Burkardt.

## **cuckoo filter plus package**

is based on the C library ``libcuckoofilter`` written by Jonah H. Harris, MIT licence, Copyright (c) 2015 Jonah H. Harris.

## **math.lerp** and **math.invlerp**

have been inspired by Luau, a fork of Lua 5.1, and a blog article written by Trys Mudford.

## **srglue** and **sragena** script-to-binary-executable utilities

have both been written by Luiz Henrique de Figueiredo for Lua 5.1 and have been adapted for Agena.

## **Debian installer files**

have been generated with the help of Gemini AI.



## **ival** interval arithmetic package

has originally been written by Luiz Henrique de Figueiredo for Lua 5.1, MIT-licenced, and has been adapted and extended for Agena. The package is a binding to the GPL-licenced ``fi_lib`` fast interval library, Version 1.2, developed by the Universities of Karlsruhe and Wuppertal.

## API access to the **ae** library for evaluating mathematical expressions in C

integrates code originally written by Luiz Henrique de Figueiredo for Lua 5.x, MIT licence.

## The **SQLite** binding

- LuaSQLite3 or Isqlite3 for short - has originally been written by Tiago Dionizio and Doug Currie for Lua 5.1 and later, Copyright (C) 2002-2016 Tiago Dionizio, Doug Currie. All rights reserved. MIT licence. The Agena source distribution includes the original SQLite 3.50.3 C code: "SQLite is in the public domain and does not require a license."

**os.getprocesses**, **os.getthreadid**, **os.getthreadseed**, **os.getusbdevices**, **os.isvalidpath**, **os.shellgeom**, **os.shellinfo**, **debug.unused**, portable `strftime()`, the **ints**, **dblhash**, **numcuckoo**, **dual** and **dd** package

have been generated or revised using Google's Gemini AI at [gemini.google.com](https://gemini.google.com), with its code put into the public domain.

## **rsorted**, **fifo** and **lifo** packages

are based on C code written by Martin Broadhurst, unknown licence.

## **mapm.xfractorial**, **mapm.xgamma**, **mapm.xpsicalc.sorted**

use Spouge approximation as presented by Kamila Szewczyk.

## **calc.eta**, **calc.horner**, **calc.invPsi**

use MIT-licenced code ported from or inspired by Julia.

## The **minizip** binding

has originally been written by Ieso-kn for Lua 5.4. MIT licence.

## The **trie** package

is based on Adam Langley's critbit C library. No licence.

## **Chaining** of relational operators

is based on code written by PlutoLang.org, Ryan Starrett, Sainan, for Pluto 0.12.0, MIT-licence.

## **tables.auto**

has originally been written by Thomas Wensch for Lua 5.x. No licence.

## **The Agena Crash Course & C Makefiles**

were redacted with the help of Google's Gemini AI to improve readability and usability.

## **The Locations Database**

used by **astro.locate** has been compiled from data available at GeoNames, Creative Commons Attribution 4.0 License.

## **NSIS 3** Windows Installer PATH Functions

have been taken from Jose Alfredo Perez' excellent documentation `Axiom Windows Installer Script`, as of April 14, 2025. The portable NSIS installer has been created with the help of Gemini AI.

---

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

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

## Table of Contents

1 Introduction .....	35
1.1 Abstract .....	35
1.2 Features .....	35
1.3 In Detail .....	36
1.4 History .....	37
1.5 Origins .....	38
2 Installing and Running Agenda .....	43
2.1 Sun Solaris 10 .....	43
2.2 Linux .....	43
2.3 Windows .....	45
2.4 OS/2 Warp 4, eComStation and ArcaOS .....	47
2.5 DOS .....	47
2.6 Mac OS X 10.5 and above .....	48
2.7 Agenda Initialisation .....	48
2.8 Installing Library Updates .....	49
3 Summary .....	53
3.1 Input Conventions in the Console Edition .....	53
3.2 Input Conventions in AgendaEdit .....	53
3.3 Getting Familiar .....	54
3.4 Useful Statements .....	55
3.5 Assignment and Unassignment .....	56
3.6 Arithmetic .....	56
3.7 Strings .....	57
3.8 Booleans .....	58
3.9 Tables .....	58
3.10 Sets .....	60
3.11 Sequences .....	60
3.12 Pairs .....	61
3.13 Conditions .....	61
3.14 Loops .....	62
3.15 Procedures .....	64
3.16 Comments .....	65
3.17 Writing, Saving, and Running Programmes .....	65
3.18 Using Packages .....	66
3.19 Printing Values .....	67
4 Data & Operations .....	71
4.1 Names, Keywords, and Tokens .....	72
4.2 Assignment .....	73
4.3 Enumeration .....	75
4.4 Deletion and the null Constant .....	75
4.5 Precedence .....	77
4.6 Arithmetic .....	77

4.6.1 Numbers .....	77
4.6.2 Arithmetic Operations .....	80
4.6.3 Increment, Decrement, Multiplication, Division .....	82
4.6.4 Mathematical Constants .....	84
4.6.5 Complex Math .....	85
4.6.6 Comparing Values .....	86
4.6.7 Range of Values .....	87
4.6.8 Adapting Basic Arithmetic Operators .....	87
4.6.9 Time Calculations .....	89
4.7 Strings .....	92
4.7.1 Representation .....	92
4.7.2 Substrings .....	93
4.7.3 Escape Sequences .....	94
4.7.4 Concatenation .....	95
4.7.5 String Operators and Functions .....	95
4.7.6 Comparing Strings .....	98
4.7.7 Patterns and Captures .....	98
4.8 Boolean Expressions .....	104
4.9 Tables .....	106
4.9.1 Arrays .....	107
4.9.2 Dictionaries .....	111
4.9.3 Table, Set and Sequence Operators .....	113
4.9.4 Table Functions .....	116
4.9.5 Table References .....	118
4.9.6 Unpacking Tables by Name .....	119
4.9.7 Defining Multiple Constants Easily .....	119
4.10 Sets .....	120
4.11 Sequences .....	123
4.12 Stack Programming .....	128
4.13 More on the create Statement .....	130
4.14 Pairs .....	130
4.15 Registers .....	133
4.16 Exploring the Internals of Structures .....	138
4.17 Other Types .....	138
 5 Control .....	 141
5.1 Conditions .....	141
5.1.1 if Statement .....	141
5.1.2 if Operator, Version One .....	144
5.1.3 if Operator, Version Two .....	145
5.1.4 Short-cut Condition with ? and -? Tokens .....	146
5.1.5 case Statement .....	147
5.1.6 case of Statement .....	148
5.2 Loops .....	149
5.2.1 while Loops .....	149
5.2.2 for/to Loops .....	153
5.2.3 for/downto Loops .....	155
5.2.4 for/in Loops over Tables .....	155

5.2.5 for/in Loops over Sequences and Registers .....	157
5.2.6 for/in Loops over Strings .....	157
5.2.7 for/in Loops over Sets .....	158
5.2.8 for/in Loops over Procedures .....	158
5.2.9 for/while and for/until Loops .....	160
5.2.10 for/as & for/until Loops .....	161
5.2.11 Loop Jump Control .....	162
5.2.12 Conditional for Loops .....	164
5.2.13 Scope I: scope and epocs .....	166
5.2.14 Scope II: with Statement .....	166
5.2.15 with Statement for Dictionaries .....	167
5.2.16 Alternative to Closing Keywords .....	168
 6 Programming .....	 171
6.1 Procedures .....	171
6.2 Local Variables .....	173
6.3 Global Variables .....	175
6.4 Changing Parameter Values .....	175
6.5 Optional Arguments .....	176
6.6 Passing Options in any Order .....	178
6.7 Type Checking .....	178
6.8 Error Handling .....	180
6.8.1 The error Function .....	180
6.8.2 Type Checks in Procedure Parameter Lists .....	181
6.8.3 Checking the Type of Return of Procedures .....	182
6.8.4 The assume Function .....	183
6.8.5 Trapping Errors with protect/lasterror .....	184
6.8.6 Trapping Errors with the try/catch Statement .....	185
6.8.7 Trapping Errors with pre and post Clauses .....	186
6.9 Multiple Returns .....	186
6.10 Procedures that Return Procedures .....	188
6.11 Shortcut Procedure Definition .....	189
6.12 User-Defined Procedure Types .....	191
6.13 Scoping Rules .....	192
6.14 Access to Loop Control Variables within Procedures .....	194
6.15 Sandboxes .....	194
6.16 Altering the Environment at Run-Time .....	195
6.17 Packages .....	197
6.17.1 Writing a New Package .....	197
6.17.2 The initialise Function .....	198
6.18 Remember Tables .....	200
6.18.1 Standard Remember Tables .....	200
6.18.2 Read-Only Remember Tables .....	202
6.18.3 Functions for Remember Table Administration .....	204
6.19 Overloading Operators with Metamethods .....	204
6.19.1 First Example: Constructor, Read & Write Operations, Prettyprinter .....	205
6.19.2 Write-Protection .....	209
6.19.3 Type-Checking .....	211

6.19.4 Calling a Structure like a Function .....	212
6.19.5 Iterating Objects in for/in Loops .....	212
6.19.6 Registering Metamethods .....	213
6.20 Memory Management, Garbage Collection, and Weak Structures .....	214
6.21 Extending Built-in Functions .....	215
6.22 Closures: Procedures that Remember their State .....	216
6.23 Self-defined Binary Operators .....	219
6.24 OOP-style Methods on Tables .....	219
6.25 Assigning Tables to Procedures .....	221
6.26 Summary on Procedures .....	223
6.27 I/O .....	223
6.27.1 Reading Text Files .....	223
6.27.2 Writing Text Files .....	224
6.27.3 Keyboard Interaction .....	225
6.27.4 Default Input, Output, and Error Streams .....	226
6.27.5 Locking Files .....	226
6.27.6 Interaction with Applications .....	226
6.27.7 CSV Files .....	227
6.27.8 XML Files & JSON Objects .....	227
6.27.9 dBASE III/IV & SQLite Files .....	227
6.27.10 INI Files .....	227
6.28 Linked Lists .....	228
6.29 Numeric C Arrays .....	231
6.30 Userdata and Lightuserdata .....	231
6.31 The Registry .....	231
6.32 Functional-Style Programming .....	233
Index .....	239

## Part One

### Primer





## Chapter One

# Introduction



# 1 Introduction

## 1.1 Abstract

Agena is a procedural programming language designed for scientific, educational, linguistic, and many other applications, including scripting.

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

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

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

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

<http://sourceforge.net/projects/adena>.

## 1.2 Features

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

Agena provides all the common functionality found in imperative languages:

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

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

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

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

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

### 1.3 In Detail

Agena offers various flow control facilities such as

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

Data types provided are:

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

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

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

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

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

Some other features are:

- graphics in the Solaris, Mac, 32-bit Linux, Raspberry Pi, and Windows editions, provided by the **gdi** package,
- IPv4 and IPv6 networking,
- functions to support fast text processing,
- configuration of user's environment via the Agena initialisation file,
- an easy-to-use package system also providing a means to both load a library and define short names for all package procedures at a stroke,
- the **binio** package to easily write and read files in binary mode,
- facility to store any data to a file and read it back later,
- undergraduate Calculus, Linear Algebra, Statistics and Combinatorics packages,
- enumeration and multiple assignment,
- scope control via the **scope/epocs** keywords,
- efficient stack programming facilities,
- bitwise operators,
- direct access to the file system,
- arbitrary precision mathematical libraries,
- dBASE, SQLite, XML, CSV, INI, ZIP, GZIP and TAR file support,
- a simple editor called AgenaEdit for Solaris, Windows, Mac OS X and Linux.

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

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

## 1.4 History

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

Plans became concrete in 2005 when I learned Lua to write procedures for phonetic analysis and also learned ANSI C to transfer them into a C package. In autumn 2006 the first modifications of the Lua parser started with extensive modifications and extensions of the lexer, parser and the Lua Virtual Machine in

summer 2007. Most of Agenda's basic functionality had been completed in March 2008, followed by the first new data structure, Cantor sets, one month later, some more data structures, and a lot of fine-tuning and testing thereafter. Finally, in January 2009, the first release of Agenda was published at Sourceforge.

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

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

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

## 1.5 Origins

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

### Maple V Release 3 and later

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

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

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

## Algol 68

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

- **case/of/esac** control

has originally been introduced with Algol 68.

## Algol 60

- **entier.**

Algol 60 is the parent of Algol 68.

## Modula-2

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

## C

- **printf**, and most of Lua's system functions,
- compound operators such like ++, etc.,
- bitwise operators: <<, >>, etc.

C actually is a descendent of Algol 68.

## COBOL

- **for/until** loops.

## Sinclair ZX Spectrum BASIC

- **clear, cls, int.**

## SQL and ABC

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

## PL/I and REXX

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

## Eiffel

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

## Ada and Perl

- inspired the **next when**, **break when** and **return when** statements.



## Chapter Two

# Installing & Running Agenda



## 2 Installing and Running Agena

### 2.1 Sun Solaris 10

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

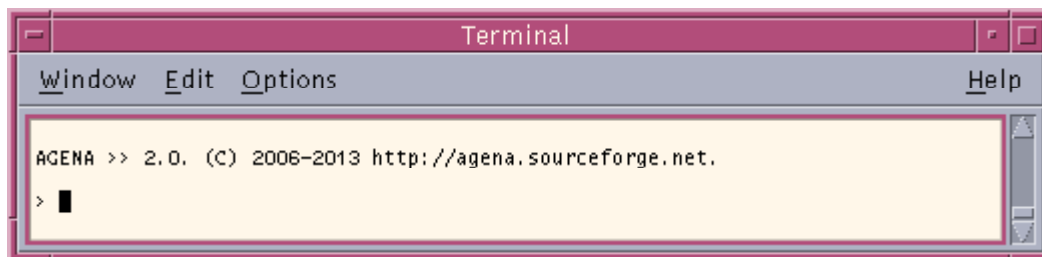
```
> gzip -d agena-x.y.z-sol10-x86-local.gz
```

Then install it with the Solaris package manager:

```
> pkgadd -d agena-x.y.z-sol10-x86-local
```

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

Make sure you have the *expat*, *fontconfig*, *freetype*, *libg2*, *libgmp-10*, *jpeg*, *libgcc*, *libgd*, *libiconv*, *libintl*, *(lib)ncurses*, *libtinfo*, *libmpfr-6*, *libpng*, *pcre-2.8*, *readline*, *(lib)xpm*, and *zlib* libraries installed. From the command line, type `adena` and press



RETURN.

Image 1: Start-up message in Solaris

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

## 2.2 Linux

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

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

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

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

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

Note that you must have the *expat*, *fontconfig*, *freetype*, *libg2*, *libgmp-10*, *libjpeg62*, *libgcc*, *libgd* (version 2.0.36 or earlier), *libiconv*, *libintl*, *libmpfr-6*, *libncurses6*, *libtinfo6*, *libpng12*, *libreadline8*, *(lib)xpm*, *libpcre2-8-0*, *zlib1g*, *libexpat1*, *x11proto-xext-dev* and *zlib* libraries installed before.

You might try to run this in your shell:

```
sudo apt install libc6 libgmp10 libmpfr6 libreadline8 libncurses6  
libpcre2-8-0 zlib1g libexpat1 libpng16-16 libjpeg62
```

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

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

The name of the Linux package is `agena`.

## 2.3 Windows

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

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

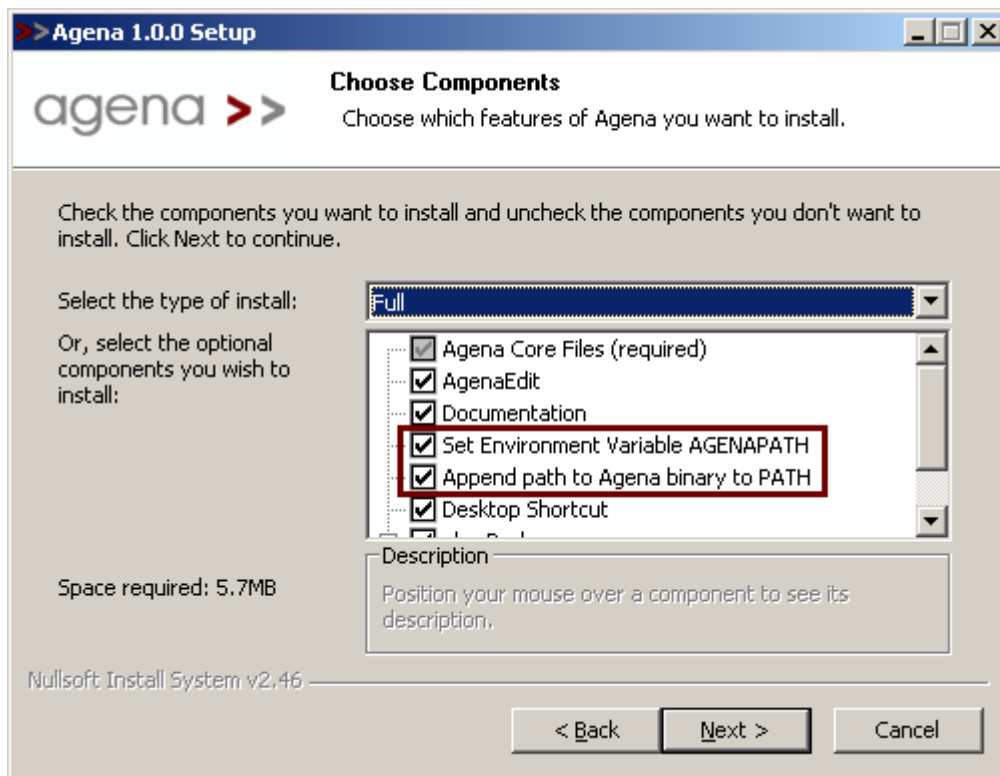


Image 2: Leave the framed settings checked

**WARNING:** If your system environment variable PATH already consists of 8,000 or more characters, do NOT select the 'Append path to Agena binary to PATH' option, as this might corrupt the PATH setting, although this should not happen with the NSIS 3 installer. Try to add the path manually instead.

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

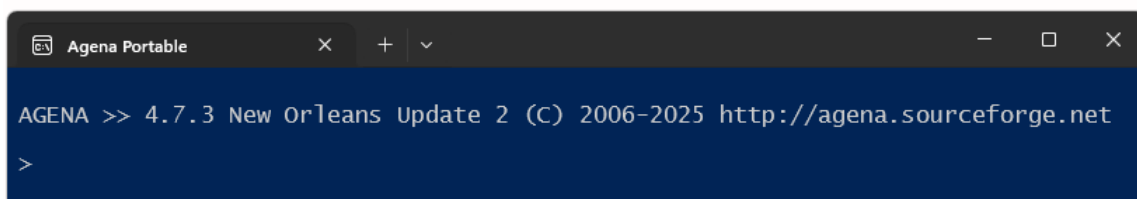


Image 3: Start-up message in Windows

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

**Hint:** On some 64-bit flavours of Windows 2003 Server and Windows 2008 Server you may need to set the `agena.exe` binary file to Windows 2000 or Windows XP compatibility mode in order for the interpreter to start successfully.

For the portable version:

In an NT shell, create a folder called ``agena`` anywhere on your drive, change into this folder and decompress this ZIP file into it preserving the subdirectory structure of the ZIP file, which should now look like this:

```
<DIR>      bin
      687,461 change.log
<DIR>      doc
<DIR>      lib
      2,114  licence
      862  readme.w32
      475  run.bat      <-- start-up batch file
<DIR>      share      <-- includes icons
```

Then create a desktop shortcut to the batch file ``run.bat`` that resides at the root of the installation. Choose an icon located in the ``share\icons`` folder to beautify the shortcut. Recommendation: ``agena256.ico`` for Windows 2000 and above (e.g. XP, Vista, and above).

Check the properties of the shortcut to be sure that the value in field "Start in" is the path to the root Agena folder, e.g. ``C:\agena``, and not to ``C:\agena\bin`` or anything else.

The 2-click installer with the ``win32-portable.exe`` extension lets you set up Agena without administrative rights and leaves your system environment variables unchanged. It installs Agena and its documentation, the AgenaEdit editor, all libraries plus data and schema files. Windows 2000 to 11 are supported.

The minimum CPU requirement for the Windows installers that have the phrase ``-sse2-`` in the file name is: Pentium 4/Athlon 64 or newer, SSE2 instruction set, AVX is not required. The SSE2-compatible version of Agena is around 1.5 percent faster than the one using the x87 instruction set. These installers have lately been deprecated due to low demand.

## 2.4 OS/2 Warp 4, eComStation and ArcaOS

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

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

```
yum install libc readline ncurses tinfo gmp pcre-2.8
```

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

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

## 2.5 DOS

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

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

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

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

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

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

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

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

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

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

## 2.6 Mac OS X 10.5 and above

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

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

Note that you may have to install the *readline* and *pcre-2.8* libraries before.

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

## 2.7 Agena Initialisation

When you start Agena, the following actions are taken:

1. The standard packages are initialised so that they become available to the user immediately.
2. All global values are copied from the **\_G** table to its copy **\_origG**, so that the **restart** function can restore the original environment if invoked.
3. The system variables **libname** and **mainlibname** pointing to the main Agena library folder and optionally to other folders is set by either querying the environment variable `AGENAPATH` or - if not set - checking whether the current working directory contains the string `/adena` or any other eligible folder name, building the path accordingly.

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

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

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



issued. If there are multiple `library.agn` files in your path, only the first one found is initialised.

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

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

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

In UNIX based systems, if both the `.agenainit` and `agenda.ini` files exist, then `.agenainit` will be read first.

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

## 2.8 Installing Library Updates

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

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

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

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



## Chapter Three

### Overview



## 3 Summary

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

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

### 3.1 Input Conventions in the Console Edition

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

- If an expression is finished with a colon, it is evaluated and its value is printed at the console.
- If the expression ends with a semicolon or neither with a colon nor a semicolon, it is evaluated, but nothing is printed on screen.

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

### 3.2 Input Conventions in AgendaEdit

The Windows, Solaris, Mac OS X and Linux distributions contain an editor providing syntax-highlighting and the facility to run the code you edited. You may start **AgendaEdit** via the Start Menu, or by typing `agendaedit` in a shell.

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

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

- Hit the F5 key to compute all statements you entered.
- Consecutive statements can be executed by selecting them and hitting the F6 key.
- To display results in the output window, pass the respective expression to the **print** function, e.g.:

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

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

The screenshot shows the AGENA 6.1.1 IDE with two windows. The main window, titled 'Untitled (modified)', contains the following code:

```

1 f := <: x -> exp(sin(x)) :>;
2
3 for x from -2 to 2 by 0.25 do
4   print(x, f(x))
5 od;
6
7 A := < < 2, 3, -1, 1 >, < 1, 3, 1, 2 >, < -2, -2, 4, 4 > >;
8 print(linalg.gausselim(A));

```

The output window, titled '(2) AGENA >> 6.1.1 (Done)', displays the results of the execution:

```

-2      0.40280712612353
-1.75   0.37381810622153
-1.5     0.36880213930276
-1.25    0.38713391223619
-1       0.43107595064559
-0.75    0.5057874485886
-0.5     0.61913896109773
-0.25    0.78082520824503
0        1
0.25     1.2806963574442
0.5      1.6151462964421
0.75     1.9771150960557
1        2.3197768247159
1.25     2.5830855122552
1.5      2.7114810176822
1.75     2.6750978172454
2        2.482577728015
[ 2, 3, -1, 1 ]
[ 0, 1, 3, 5 ]
[ 0, 0, -3, -6 ]      3      6

```

At the bottom of the output window are buttons for 'Break', 'Restart', and 'Close'.

### 3.3 Getting Familiar

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

```

> print(1+2)
3

```

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

```

> c := 25;

```

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

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

```

> print(1.8*c + 32);
77

```

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

```

> print(arcsin(1));
1.5707963267949

```

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

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

### 3.4 Useful Statements

Instead of using **print**, you may also output results by entering an expression and completing it with a colon - this also works with expressions spread across multiple lines:

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

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

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

```
> ans:
1.8378770664093
```

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

```
> cls
```

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

```
> restart
```

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

```
> bye
```

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

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

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

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

```
Agena$ restart;
```

All the statements that you enter on the command-line and that are syntactically correct can be written to a file named `agenda.log` in your current working directory by switching on the logging feature:

```
> history.on()
```

You can switch off logging with:

```
> history.off()
```

Alternatively, to write the logfile to another place, issue for example:

```
> history.on('c:/agenda/log.txt')
```

By default, there is no command-line logging, that means every time you start a new Agena session, you must explicitly turn it on.

### 3.5 Assignment and Unassignment

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

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

Variables can be deleted by assigning **null** or using the **clear** statement. The latter also immediately performs a garbage collection. Note the usage of the colon to print results easily.

```
> a := null:  
null
```

```
> clear a;
```

```
> a:  
null
```

### 3.6 Arithmetic

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

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



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

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

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

For polar notation use the **!!** operator:

```
> sqrt(2)!!(Pi/4):
1+I
```

The number of digits used in the output of numbers on the Agena prompt can be controlled by the **Digits** environment variable, the default is 14:

```
> Pi:
3.1415926535898
```

```
> Digits := 17;
```

```
> Pi:
3.1415926535897931
```

```
> Digits := 5;
```

```
> Pi:
3.1416
```

Note that the setting does not control the precision of a computation, but only the output created by the pretty-printer.

### 3.7 Strings

A text can be put in single or double quotes:

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

Substrings are extracted by passing an index or index range:

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

Concatenation, search, and replacement:

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

```
> strings.instr(str, 'another'):
14
```

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

There are various other string operators and functions available.

### 3.8 Booleans

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

```
> 1 < 2:
true

> true or false:
true
```

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

### 3.9 Tables

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

Let us start with a simple example and define a table including numbers, a complex number, a string and some Booleans:

```
> tbl := [10, 20, 30, 1!1, 'a', true, false];
```

Read values in the table by indexing it. Get the first, fourth and sixth entry:

```
> tbl[1], tbl[4], tbl[6]:
10      1+I      true
```

Lets change a value in the table:

```
> tbl[4] := 40;

> tbl:
[10, 20, 30, 40, a, true, false]
```

And now we delete the fourth entry:

```
> tbl[4] := null

> tbl:
[1 ~ 10, 2 ~ 20, 3 ~ 30, 5 ~ a, 6 ~ true, 7 ~ false]
```

Check it:

```
> tbl[4]:  
null
```

Now let us create a table of tables:

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

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

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

You can get a subtable by providing a range with a lower and an upper bound, here 2 and 3, respectively:

```
> tbl[2 to 3]:  
[2 ~ [b, 7.7], 3 ~ [c, 7.59]]
```

The **insert** statement adds further values into a table, to its end.

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

Alternatively, values may be added by indexing:

```
> tbl[5] := ['e', 8.04];  
  
> tbl:  
[[a, 7.71], [b, 7.7], [c, 7.59], [d, 8.01], [e, 8.04]]
```

Of course, values can be replaced,

```
> tbl[3] := ['z', -5];  
  
> tbl:  
[[a, 7.71], [b, 7.7], [z, -5], [d, 8.01], [e, 8.04]]
```

and deleted. For example, to remove the fifth entry from tbl, ['e', 8.04], issue:

```
> tbl[5] := null;  
  
> tbl:  
[[a, 7.71], [b, 7.7], [z, -5], [d, 8.01]]
```

Alternatively use the **delete** statement which purges values:

```
> delete ['b', 7.7] from tbl;

tbl:
[1 ~ [a, 7.71], 3 ~ [z, -5], 4 ~ [d, 8.01]]
```

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

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

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

### 3.10 Sets

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

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

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

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

> s['daisy']:
false

> 'donald' in s:
true
```

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

```
> insert 'daisy' into s;

> delete 'donald' from s;

> s:
{daisy, mickey}
```

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

### 3.11 Sequences

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

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

> s[2]:
1

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

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

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

### 3.12 Pairs

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

```
> p := 10:11;

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

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

### 3.13 Conditions

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

```
> if 1 < 2 then
>   print('valid')
> elif 1 = 2 then
>   print('invalid')

> else
>   print('invalid, too')
> fi;
valid
```

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

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

```
> c := 'agenda';

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

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

```
> v := 1;

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

### 3.14 Loops

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

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

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

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

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

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

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

> c:
987
```

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

```
> c := 0;

> do
>   inc c
> as c < 10;
```

```
> c:
10
```

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

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

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

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

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

which of course is equivalent to

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

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

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

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

### 3.15 Procedures

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

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

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

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

> fact(10):
3628800
```

A procedure can call itself.

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

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

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

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

Alternatively, you can use the **def** or the **define** statement to define a procedure; for example, a function with two arguments can be defined as follows:

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

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

or alternatively (the parameters can also be given in brackets),

```
> sum := define x, y -> x + y;

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

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



### 3.16 Comments

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

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

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

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

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

Alternatively, C comments are supported, as well:

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

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

### 3.17 Writing, Saving, and Running Programmes

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

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

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

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

or in DOS-based systems:

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

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

in DOS-based systems.

---

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

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

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

in UNIX or

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

in Windows. See Appendix A5.6 for further switches.

### 3.18 Using Packages

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

For example, the kiss package provides Fast Fourier Transformation functions:

```
> import kiss;
> kiss.nextsize(7):
8
```

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

```
> import kiss alias
> nextsize(7):
8
```

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

```
> import kiss alias nextsize, factor
```

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

```
> import kiss as k;
> a := k.nextsize(7);
```

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

If you want to have detailed information on how a package is being initialised, just issue

```
> environ.kernel(debug = true)
```

and then run the **import** statement. Examples:

```
> import ads
```

Processing library: ads.

ads is an external (plus) package.

Checking path C:\agenda\src.

Checking C library file C:\agenda\src\ads.dll.

C:\agenda\src\ads.dll not present.

Checking agn library file C:\agenda\src\ads.agn: not present.

Checking path c:/agenda/lib.

Checking C library file c:/agenda/lib/ads.dll.

c:/agenda/lib/ads.dll successfully initialised.

Checking agn library file c:/agenda/lib/ads.agn: found.

All successful, now registering ads.

```
> import math
```

Processing library: math.

math is a standard library.

Nothing to be done.

### 3.19 Printing Values

We already used the **print** function to write values - numbers, strings, Booleans, tables, etc. to the screen:

```
> print('sqrt(', 2, ') = ', sqrt(2)):
sqrt(    2    ) =    1.4142135623731
```

```
> print('sqrt(' & 2 & ') = ' & sqrt(2)):
sqrt(2) = 1.4142135623731
```

The **printf** function, however, gives more control on the output format. In the following example **%d** depicts an integer and **%f** a float.

```
> printf('sqrt(%d) = %f', 2, sqrt(2)):
sqrt(2) = 1.414214
```

To print 10 decimal (fractional) places of  $\sqrt{2}$ , we put **.10** in front of the **f** specifier:

```
> printf('sqrt(%d) = %.10f', 2, sqrt(2)):
sqrt(2) = 1.4142135624
```

Next, we print  $\sqrt{2}$  with a total of 12 places (pre-decimal places plus the decimal dot plus the fractional places):

```
> printf('sqrt(%d) = %12.10f', 2, sqrt(2)):
sqrt(2) = 1.4142135624
```

The `%s` formatter represents a string:

```
> printf('%s(%d) = %18.15f', 'sqrt', 2, sqrt(2)):
sqrt(2) = 1.414213562373095
```

In the next example, we print the string `'sqrt'` with a total of ten characters and  $\sqrt{2}$  with 18 places including the decimal dot and a leading zero, right-justified.

```
> printf('%10s(%d) = %018.15f', 'sqrt', 2, sqrt(2)):
      sqrt(2) = 01.414213562373095
```

`%d` depicts any number, string or Boolean:

```
> printf('%a(%a) = %a', 'sqrt', 2, sqrt(2)):
sqrt(2) = 1.4142135623731
```

For more information and examples, check the descriptions of **printf** in Chapter 8 and **strings.format** in Chapter 9. You might also check Chapter 12 on input and output to the screen or to a file.

See also the **Digits** environment variable described in Chapter 3.6.

## Chapter Four

# Data & Operations



## 4 Data & Operations

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

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

Table 1: Available types

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

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

```
> type(0):
number

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

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

## 4.1 Names, Keywords, and Tokens

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

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

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

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

Table 2: Examples for valid and invalid names

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

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

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

The following symbols denote other tokens:

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



## 4.2 Assignment

Values can be assigned to names in the following fashions:

$$[\text{constant}] \text{ name } := \text{ value}$$

$$[\text{constant}] \text{ name}_1, \dots, [\text{constant}] \text{ name}_k := \text{value}_1, \dots, \text{value}_k$$

$$[\text{constant}] \text{ name}_1, \dots, [\text{constant}] \text{ name}_k \rightarrow \text{value}$$

In the first form, one value is stored in one variable, whereas in the second form, called 'multiple assignment statement',  $\text{name}_1$  is set to  $\text{value}_1$ ,  $\text{name}_2$  is assigned  $\text{value}_2$ , etc. In the third form, called the 'shortcut multiple assignment statement', a single value is set to each name to the left of the  $\rightarrow$  token.

First steps:

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

```
> a:
1
```

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

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

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

Multiple assignments:

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

```
> a:
1
```

```
> b:
2
```

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

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

```
> c:
1
```

```
> d:
null
```

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

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

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

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

A shortcut multiple assignment statement:

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

> x:
2.718281828459

> y:
2.718281828459
```

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

```
> constant a := 1;

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

You can declare multiple constants at a time:

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

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

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

You can mix ordinary and constant declarations:

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

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

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

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

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

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

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

### 4.3 Enumeration

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

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

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

In the first form, *name*<sub>1</sub>, *name*<sub>2</sub>, etc. are enumerated starting with the numeric value 1.

```
> enum ONE, TWO;

> ONE:
1

> TWO:
2
```

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

```
> enum THREE, FOUR from 3

> THREE:
3

> FOUR:
4
```

### 4.4 Deletion and the null Constant

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

```
clear name1 [, name2, ..., namek]
```

```
> a := 1;

> clear a;

> a:
null
```

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

```
> b := 1;

> b := null:
null
```

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

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

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

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

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

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

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

```
> assigned(v):
false

> unassigned(v):
true
```

## 4.5 Precedence

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

```

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

```

! and all self-defined binary operators and unary operators including ~~

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

```

> 1+3*4:
13

```

```

> (1+3)*4:
16

```

## 4.6 Arithmetic

### 4.6.1 Numbers

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

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

- 1
- -20
- 0
- +4

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

- -1.12
- 0.1
- .1

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

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

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

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

```
> 1e4:
10000
```

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

If a number ends in the letter `k`, `M`, `G`, `T` or `D`, then the number will be multiplied by 1,024, 1,048,576 (= 1,024<sup>2</sup>), 1,073,741,824 (= 1,024<sup>3</sup>), 1,099,511,627,776 (= 1,024<sup>4</sup>), or 12, respectively. If a number ends in the letter `k`, `m`, `g` or `t`, then the number will be multiplied by 1,000, 1,000,000, 1,000,000,000, or 1,000,000,000,000 respectively.

```
> 2k:
2000
```

```
> 1M:
1048576
```

```
> 12D:
144
```

If a number is appended by `p`, it will be converted to percentage. Furthermore, if a number literal is suffixed by the letter `d`, the number is assumed to be in degrees and automatically converted to radians. If the number is suffixed by the letter `r`, then the number is assumed to be in radians and automatically converted to decimal degrees.

```
> 50p:
0.5
```

```
> 90d:
1.5707963267949
```

```
> 1.5707963267949r:
90
```

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

System	Syntax	Examples (to decimal)
binary (integer)	0b<binary number> Or 0B<binary number>	0b10 = 2
binary (float)	0b<int>.<frac> Or 0b<int>p<int> Or 0B<int>.<frac>P<int>	0b1111.1 = 15.5
octal (integer)	0o<octal number> Or 0O<octal number>	0o10 = 8
octal (float)	0o<int>.<frac> Or 0o<int>p<int> Or 0O<int>.<frac>P<int>	0o0.04 = 0.0625
hexadecimal (integer)	0x<hexadecimal number> Or 0X<hexadecimal number>	0xa = 10
hexadecimal (float)	0x<int>.<frac> Or 0x<int>p<int> Or 0x<int>.<frac>P<int>	0x0.1 = 0.0625 0xa23p-4 = 162.1875 0X1.921FB54442D18P+1 = 3.1415926535898

If a numeric constant should be too big - i.e. out-of-range - then Agena will *not* throw an error. You can, however, let Agena validate constants by activating the appropriate check which will result in a syntax error if a constant is out-of-bounds:

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

Alternatively, you may pass the -B switch at startup on the command-line.

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

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

The **mapm** package can be used in such situations as it provides arbitrary precision arithmetic in both the real and complex domain. See Chapter 11.3 for more information.

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

### 4.6.2 Arithmetic Operations

Agena has the following arithmetical operators:

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

Table 3: Arithmetic operators

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

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

The mathematical procedures that reside in packages must always be entered by passing the name of the package followed by a dot and the name of the procedure.

Unary operators<sup>2</sup> like **ln**, **exp**, etc. can be entered with or without simple brackets.

Procedure	Operation	Library	Example and result
<b>sin</b> (x)	Sine (x in radians)	Kernel	sin(0) » 0
<b>cos</b> (x)	Cosine (x in radians)	Kernel	cos(0) » 1
<b>tan</b> (x)	Tangent (x in radians)	Kernel	tan(1) » 1.557407..
<b>sec</b> (x)	Secant	Base	sec(0) » 1
<b>csc</b> (x)	Cosecant	Base	csc(1) » 1.188395..
<b>cot</b> (x)	Cotangent	Base	cot(0.5) » 1.830487..
<b>arcsin</b> (x)	Inverse sine (x in radians)	Kernel	arcsin(0) » 0
<b>arccos</b> (x)	Arc cosine (x in radians)	Kernel	arccos(0) » 1.570796..
<b>arctan</b> (x)	Arc tangent (x in radians)	Kernel	arctan(Pi) » 1.262627..
<b>sinh</b> (x)	Hyperbolic sine	Kernel	sinh(0) » 0
<b>cosh</b> (x)	Hyperbolic cosine	Kernel	cosh(0) » 1
<b>tanh</b> (x)	Hyperbolic tangent	Kernel	tanh(0) » 0

<sup>2</sup> See Appendix A1 for a list of all unary operators.



Procedure	Operation	Library	Example and result
<b>arcsinh</b> (x)	Inverse hyperbolic sine	Kernel	<code>arcsinh(1)</code> » 0.88137..
<b>arccosh</b> (x)	Inverse hyperbolic cosine	Kernel	<code>arccosh(1)</code> » 0
<b>arctanh</b> (x)	Inverse hyperbolic tangent	Kernel	<code>arctanh(1)</code> » 0.78539..
<b>sinc</b> (x)	Cardinal sine	Base	<code>sinc(1)</code> » 0.841470
<b>cosc</b> (x)	Cardinal cosine	Base	<code>cosc(1)</code> » 0.540302
<b>tanc</b> (x)	Cardinal tangent	Base	<code>tanc(1)</code> » 1.557408
<b>exp</b> (x)	Exponentiation $e^x$	Kernel	<code>exp(0)</code> » 1
<b>ln</b> (x)	Natural logarithm	Kernel	<code>ln(1)</code> » 0
<b>log</b> (x, b)	Logarithm of x to the base b	Kernel	<code>log(8, 2)</code> » 3
<b>sqr</b> t(x)	Square root of x	Kernel	<code>sqr</code> t(2) » 1.414213..
<b>cbr</b> t(x)	Cubic root of x	Base	<code>cbr</code> t(2) » 1.259921..
<b>root</b> (x, n)	Non-principal n-th root of x	Base	<code>root(2, 3)</code> » 1.259921..
<b>proot</b> (x, n)	Principal n-th root of x	Base	<code>proot(2, 3)</code> » 1.259921..
<b>hypot</b> (x, y)	Hypotenuse	Base	<code>hypot(1, 2)</code> » 2.2360..
<b>gamma</b> (x)	$\Gamma$ x	Base	<code>gamma(4)</code> » 6
<b>lngamma</b> (x)	$\ln \Gamma$ x	Kernel	<code>exp(lngamma(3+1))</code> » 6
<b>fact</b> (n)	Factorial	Base	<code>fact(3)</code> » 6
<b>erf</b> (x)	Error function	Base	<code>erf(1)</code> » 0.84270..
<b>abs</b> (x)	Absolute value of x	Kernel	<code>abs(-1)</code> » 1
<b>sign</b> (x)	Sign of x	Kernel	<code>sign(-1)</code> » -1
<b>entier</b> (x) <b>floor</b> (x)	Round x downwards to the nearest integer	Kernel	<code>entier(2.9)</code> » 2 <code>entier(-2.9)</code> » -3
<b>ceil</b> (x)	Rounds x upwards to the nearest integer	Base	<code>ceil(2.9)</code> » 3 <code>ceil(-2.9)</code> » -2
<b>int</b> (x)	Rounds x to the nearest integer towards zero	Kernel	<code>int(2.9)</code> » 2 <code>int(-2.9)</code> » -2
<b>frac</b> (x)	Fractional part	Base	<code>frac(-Pi)</code> » -0.141592..
<b>round</b> (x, d)	Rounds the real value x to the d-th digit	Base	<code>round(sqrt(2), 2)</code> » 1.41
<b>even</b> (x)	Checks whether x is even	Kernel	<code>even(2)</code> » true
<b>odd</b> (x)	Checks whether x is odd	Kernel	<code>odd(2)</code> » false
<b>sumup</b> ([...])	Sum	Kernel	<code>sumup([1, 2, 3])</code> » 6
<b>mean</b> ([...])	Arithmetic mean	stats	<code>stats.mean([1, 2, 3])</code> » 2
<b>gmean</b> ([...])	Geometric mean	stats	<code>stats.gmean([1, 2, 3])</code> » 2.16
<b>hmean</b> ([...])	Harmonic mean	stats	<code>stats.hmean([1, 2, 3])</code> » 1.636
<b>median</b> ([...])	Median	stats	<code>stats.median([1, 2, 3, 4])</code> » 2.5
<b>sd</b> ([...])	Standard deviation	stats	<code>stats.sd([1, 2, 3, 4])</code> » 1.12
<b>ad</b> ([...])	Absolute deviation	stats	<code>stats.ad([1, 2, 3, 4])</code> » 1

Table 4: Common mathematical functions

In addition, Agena can conduct bitwise operations on numbers.

Operator	Operation	Details / Example
<b>&amp;&amp;</b>	Bitwise `and`	<code>7 &amp;&amp; 2</code> » 2
<b>  </b>	Bitwise `or`	<code>1    2</code> » 3
<b>^^</b>	Bitwise `exclusive-or`	<code>7 ^^ 2</code> » 5
<b>~~</b>	Bitwise complement (bitwise `not`)	<code>~~7</code> » -8
<b>&lt;&lt;, &gt;&gt;</b>	Bitwise shift	<code>&lt;&lt;</code> conducts a left-shift (multiplication with 2), <code>&gt;&gt;</code> a right-shift (division by 2).
<b>&lt;&lt;&lt;, &gt;&gt;&gt;</b>	Bitwise rotation	<code>&lt;&lt;&lt;</code> and <code>&gt;&gt;&gt;</code> rotate bits left- and rightwards.
<b>nand</b>	bitwise complement `and`	Equivalent to <code>~~(a &amp;&amp; b)</code> .
<b>nor</b>	bitwise complement `or`	Equivalent to <code>~~(a    b)</code>
<b>xnor</b>	bitwise complement exclusive-`or`	Equivalent to <code>~~(a ^^ b)</code>
<b>getbit getbits</b>	returns stored bit(s)	<code>getbit(3, 1)</code> , <code>getbits(3)</code>
<b>setbit setbits</b>	sets bit(s)	<code>setbit(3, 1)</code> » 1, <code>setbits(8, reg(1, 0, 0))</code> » 12

Table 5: Bitwise operators and functions

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

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

The default is restored as follows:

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

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

### 4.6.3 Increment, Decrement, Multiplication, Division

Instead of incrementing or decrementing a value, say

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

by entering a statement like

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

2

you can use the **inc** and **dec** commands<sup>3</sup> which are also around 10% faster:

```
inc name [, value];
dec name [, value];
```

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

```
> inc a;

> a:
3

> dec a;

> a:
2

> inc a, 2;

> a:
4
> dec a, 3;

> a:
1
```

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

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

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

Operator	Operation	Equivalent
<code>+=</code>	addition	<b>inc</b> statement
<code>-=</code>	subtraction	<b>dec</b> statement
<code>*=</code>	multiplication	<b>mul</b> statement
<code>/=</code>	division	<b>div</b> statement
<code>\=</code>	integer division	<b>intdiv</b> statement
<code>%=</code>	modulo	<b>mod</b> statement
<code>&amp;=</code>	string concatenation	n/a
<code>@=</code>	conditional multiplication	n/a

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

---

<sup>3</sup> Finishing an **inc** or **dec** statement with a colon instead of a semicolon is refused.

```
> a:
4
```

For bitwise operations, we have the following mutate operators:

Operator	Operation
<code>&amp;&amp;:=</code>	bitwise AND
<code>  :=</code>	bitwise OR
<code>^^:=</code>	bitwise XOR
<code>&lt;&lt;:=</code>	arithmetic left-shift
<code>&gt;&gt;:=</code>	arithmetic right-shift
<code>&lt;&lt;&lt;:=</code>	rotational left-shift
<code>&gt;&gt;&gt;:=</code>	arithmetic right-shift

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

```
> c := 0;

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

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

> c++;          # used as a statement

> print(c)
2
```

#### 4.6.4 Mathematical Constants

Agena features arithmetic constants mentioned in Appendix A3. All mathematical constants are protected and cannot be changed.

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

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

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

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

The **fractional** operator checks whether a value has a fractional part and thus is not an integer.

```
> fractional(1):
false
```

```
> fractional(1.1):
true
```

## 4.6.5 Complex Math

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

```
> a := 1!1;

> b := 2+3*I;

> a+b:
3+4*I

> a*b:
-1+5*I

> sqrt(2)!!(Pi/4):
1+I
```

The following operators work on rational numbers as well as complex values: **+**, **-**, **\***, **/**, **^**, **\*\***, **=**, **~=**, **<>**, **abs**, **arccos**, **arcsec**, **arcsin**, **arctan**, **conjugate**, **cos**, **cosh**, **entier**, **exp**, **flip**, **imag**, **invsqrt**, **lngamma**, **ln**, **log**, **real**, **sign**, **sin**, **sinh**, **sqrt**, **tan**, **tanh**, and unary minus. With these operators, you can also mix numbers and complex numbers in expressions.

You will find that most mathematical functions are also applicable to complex values.

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

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

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

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

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

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

#### 4.6.6 Comparing Values

Relational operators can compare both numeric and complex values. Whereas all relational operators work on numbers, complex numbers can only be compared for equality (=) or inequality (<>), or approximate equality (~=) or inequality (~<>).

You can also compare numbers with complex numbers with the =, ~=, <> and ~<> operators.

Operator	Description	Complex value support and mixed comparison
<	less than	no
>	greater than	no
<=	less than or equals	no
>=	greater than or equals	no
=	equals	yes
~=	approximately equals	yes
<>	not equals	yes
~<>	approximately not equals	yes
in	in range	no

```
> 1 < 2:
true
```

```
> 1 = 1:
true
```

```
> 1 <> 1:
false
```

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

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

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

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

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

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

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

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

To determine whether a number is part of a closed interval, use the **in** operator or just chain relational operators:

```
> 0 <= 2 <= 10, 2 in 0 : 10:
true      true
```

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

```
> 1 in +++1:---10:
false
```

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

#### 4.6.7 Range of Values

The following ranges apply to Agena numbers and complex numbers:

Characteristic	Value
smallest representable number	$-1.797\ 693\ 134\ 862\ 315 \times 10^{308}$
largest representable number	$+1.797\ 693\ 134\ 862\ 315 \times 10^{308}$
largest positive integer without loss of precision	$9.007199254741 \times 10^{15} = 2^{53}$
smallest subnormal (negative) positive number	$(-)4.9406564584124654\text{e-}324$
largest subnormal (negative) positive number	$(-)2.2250738585072009\text{e-}308$

#### 4.6.8 Adapting Basic Arithmetic Operators

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

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

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

Examples:

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

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

> 1 div 0:
division by zero
```

A multiplication handler:

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

> 1e308 mul 1e308:
overflow
```

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

```
> math.add := proc(a, b, kind) is
>   case kind
>     of 0b0000 then return a + b # no exception
>     # very large value to be added to (subtracted) value close to 0:
```



```
>         of 0b0010 then return a + b
>         # very large values to be added (or subtracted):
>         of 0b0100 then error('overflow')
>         # both operands are close to zero:
>         of 0b1000 then return a + b
>     esac
> end;
```

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

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

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

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

```
> 1e308 inc 1e308:
overflow

> environ.arithstate():
1
```

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

#### 4.6.9 Time Calculations

Agenda has various functions that allow to compute with dates and times. They are included in the **math**, **clock** and **astro** packages.

Let us look at some of their features: Probably the most easy-to-use time function is **math.dd** which converts a number in Degrees, Minutes, Seconds (DMS) notation to a decimal number, for example 1 hour, 30 minutes and 45 seconds:

```
> a := math.dd(1.3045):
1.5125
```

1 hour, 29 minutes and 16 seconds:

```
> b := math.dd(1.2916):
1.4877777777778
```

Add a and b, resulting in a decimal result,

```
> a + b:
3.0002777777778
```

and convert the sum back to DMS notation representing 3 hours, 0 minutes, 1 second:

```
> math.dms(ans):
3.0001
```

Split the result, the DMS number 3.0001 we computed with the previous statement, into its components degrees, minutes and seconds:

```
> math.splitdms(ans):
3      0      1
```

Convert degrees, minutes and seconds to decimal representation:

```
> math.todecimal(3, 0, 1):
3.0002777777778
```

Split the decimal representation back into degrees, minutes and seconds:

```
> math.tosgesim(3.0002777777778):
3      0      1
```

The **clock** package provides sophisticated functions that cope with much larger time values, preventing round-off errors as much as possible. Load the short forms:

```
> import clock alias
```

Subtract 10 hours and fifteen minutes from 20 hours and 15 minutes:

```
> tm(20, 15, 0) - tm(10, 15, 0):
tm(10, 0, 0)
```

61 seconds are automatically converted to 1 minute and 1 second:

```
> tm(0, 61):
tm(0, 1, 1)
```

Check Chapter 11.8 for further details.

The **os** package has functions to retrieve the current time and that also do some basic calendar calculations. To get the current time as a timestamp, enter:

```
> os.date():
2025/08/24 16:36:11.286
```

Do the same but compute the number of seconds elapsed since January 01, 1970.

```
> os.time(): # seconds plus milliseconds
1756046178    286
```

For another date pass year, month, day and optionally time, for example for noon January 01, 2000:

```
> os.time([2000, 1, 1, 12, 0, 0]):
946724400
```

To convert this back into a timestamp, you may issue:

```
> os.date('%c', 946724400):
01/01/00 12:00:00
```

The function supports of a lot specifiers to extract details:

```
> os.date('%A, %d %b %Y %H:%M:%S, %z', 946724400):
Saturday, 01 Jan 2000 12:00:00, W. Europe Standard Time
```

**os.now** determines a wealth of information - you can also call it without argument for the current time:

```
> os.now(946724400):
[dst ~ false, gmt ~ [2000, 1, 1, 11, 0, 0, 6, 0, 0, AM, January, Saturday],
 jd ~ 2451545, localtime ~ [2000, 1, 1, 12, 0, 0, 6, 0, 0, PM, January,
 Saturday], lsd ~ 36527.5, mseconds ~ 339, seconds ~ 946724400, td ~ 60,
 tz ~ 60]
```

Alternatively pass a table of the year, month, day and optionally hour, minute, second of interest:

```
> os.now([2000, 1, 1, 12, 0, 0]):
[dst ~ false, gmt ~ [2000, 1, 1, 11, 0, 0, 6, 0, 0, AM, January, Saturday],
 jd ~ 2451545, localtime ~ [2000, 1, 1, 12, 0, 0, 6, 0, 0, PM, January,
 Saturday], lsd ~ 36527.5, mseconds ~ 751, seconds ~ 946724400, td ~ 60,
 tz ~ 60]
```

Similarly, **os.date** can retrieve some details, too,

```
> os.date('*t', [2000, 12, 31, 12, 0, 0]):
[day ~ 31, hour ~ 12, isdst ~ false, min ~ 0, month ~ 12, msec ~ 847,
 sec ~ 0, wday ~ 7, yday ~ 366, year ~ 2000]
```

and now for UTC:

```
> os.date('!*t', [2000, 12, 31, 12, 0, 0]):
[day ~ 31, hour ~ 11, isdst ~ false, min ~ 0, month ~ 12, msec ~ 391,
 sec ~ 0, wday ~ 7, yday ~ 366, year ~ 2000]
```

The **astro** package has calendar and some astronomical functions. To convert a Gregorian date into a Julian date (J2000.0), type:

```
> import astro

> astro.jdate(2000, 1, 1, 12, 0, 0):
2451545
```

and convert back:

```
> astro.cdate(2451545):
2000      1      1      0.5      12      0      0
```

See Chapter 11.9 for further information.

## 4.7 Strings

### 4.7.1 Representation

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

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

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

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

> str:
I am a string.
```

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

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

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

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

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

which prints as

```
> str:
Two
lines
```

If you do not want to include line breaks and succeeding white spaces, use the `\z` escape sequence:

```
> str := '\z
>      1234\z
>      5678'
```

```
> str:
12345678
```

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

```
> '':
```

### 4.7.2 Substrings

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

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

> str[1];
I
```

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

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

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

```
> str := 'string'

> str[3]:
r

> str[3 to 5]:
rin

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

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

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

> str[3 to -2]:
rin

> str[-3 to -2]:
in
```

```
> str[-3]:
i
```

If the right index is greater than the length of string the string, it is auto-corrected to the string length. If the left border is 0 or greater than the length of the string, however, an error will be returned.

### 4.7.3 Escape Sequences

In Agena, a text can include any escape sequences<sup>4</sup> known from ANSI C, e.g.:

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

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

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

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

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

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

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

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

Likewise, a backslash is represented by typing it twice. See Appendix A8 for more escape sequences.

A string can also be enclosed in backquotes. In this case, there will be no escaping which is especially useful when working with regular expressions:

```
> `\\n`:
\\n
```

The only exception is the escape sequence `\q` which exists in backquoted strings only and represents a backquote itself:

---

<sup>4</sup> See also Appendix A8.

```
> `qhallo`q`:  
`hallo`
```

#### 4.7.4 Concatenation

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

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

Numbers (also complex ones) and Booleans are supported, as well, so you do not need to convert them with the **tostring** function before applying `&`. The operator also treats **null** like the empty string.

```
> 1 & ' duck':  
1 duck  
  
> 1-Pi*I & ' is a complex number':  
1-3.1415926535898*I is a complex number
```

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

```
> a := 'In';  
  
> a &:= 'Sight';  
  
> a:  
InSight
```

#### 4.7.5 String Operators and Functions

Agenda has basic operators useful for text processing:

Operator	Return	Function
<code>s in t</code>	number or <b>null</b>	Checks whether a substring <code>s</code> is included in string <code>t</code> . If true, the position of the first occurrence of <code>s</code> in <code>t</code> will be returned; otherwise <b>null</b> will be returned.
<code>s atendof t</code>	number or <b>null</b>	Checks whether a string <code>t</code> ends in a substring <code>s</code> . If true, the position of the position of <code>s</code> in <code>t</code> will be returned; otherwise <b>null</b> will be returned.
<code>replace(s, p, r)</code>	string	Replaces all patterns <code>p</code> in string <code>s</code> with substring <code>r</code> . If <code>p</code> is not in <code>s</code> , then <code>s</code> will be returned unchanged. <code>p</code> might also be the position (a positive integer) of the character to be replaced.
<code>s split d</code>	sequence of strings	Splits a string into its words with <code>d</code> as the delimiting character(s). The items are returned as a sequence of strings.

Operator	Return	Function
<b>size(s)</b>	number	Returns the length of string s. If s is the empty string, 0 will be returned.
<b>abs(s)</b>	number	Returns the numeric ASCII code of character s.
<b>char(n)</b>	string	Returns the character corresponding to the given numeric ASCII code n.
<b>lower(s)</b>	string	Converts a string to lowercase. Western European diacritics are recognised.
<b>upper(s)</b>	string	Converts a string to uppercase. Western European diacritics are recognised.
<b>tonumber(s)</b>	number or complex value	Converts a string into a number or complex number.
<b>tostring(n)</b>	string	Converts a number to one string. If a complex value is passed, the real and imaginary parts are returned separately as two strings.
<b>trim(s)</b>	string	Deletes leading and trailing spaces as well as excess embedded spaces.

Table 7: String operators

Some examples:

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

The character `s` is at the third position:

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

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

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

`str` is eight characters long:

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

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

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

translated back to

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



Put all characters in `str` to uppercase:

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

And now the reverse:

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

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

Function	Functionality	Example
<b>in</b>	Returns the first position of a substring (left operand) in a string (right operand); if the substring cannot be found, it returns <b>null</b> .	'tr' in 'string' » 2
<b>instr</b>	<p>Looks for the first match of a pattern (second argument) in a string (first argument). If it finds a match, then <b>instr</b> returns its position; otherwise, it returns <b>null</b>. An optional numerical argument specifies where to start the search. The function supports pattern matching, almost similar to regular expressions. The function is more than twice as fast as <b>strings.find</b>. If <b>true</b> is given as a fourth argument, pattern matching is switched off to speed up the search.</p> <p>If the option 'reverse' is given, then starting from the right end and always running to its left beginning, the function looks for the first match of the substring and returns the position where the pattern starts with respect to its left beginning. When searching from right to left, pattern matching is not supported.</p>	<pre>instr(   'agena',   '[aA]g',   1) » 1</pre> <pre>instr('agena', 'a',   'reverse') » 5</pre>
<b>atendof</b>	Checks whether a string (right operand) ends in a substring (left operand). If true, the position will be returned; otherwise <b>null</b> will be returned.	'ing' atendof 'raining' » 5
<b>strings.find</b>	Returns the first match of a substring (second argument) in a string (first argument) and returns the positions where the pattern starts <i>and ends</i> . An optional third argument specifies the position where to start the search. If it does not find a pattern, the function returns <b>null</b> .	<pre>strings.find(   'string', 'tr') » 2, 3</pre> <pre>strings.find(   'string', 'tr',   3) » null</pre>

Function	Functionality	Example
	<p>The function supports pattern matching facilities described in Chapter 9.1.3.</p> <p>See also: <b>strings.mfind</b>, which returns all occurrences.</p>	<pre>strings.find(   'string', 't.') » 2, 3</pre>
<b>replace</b>	<p>In a string (first argument) replaces all occurrences of a substring (second argument) with another one (third argument) and returns a new string. Pattern matching facilities are not supported.</p> <p>A sequence of replacement pairs can be passed to the function, too.</p> <p>See also <b>strings.gsub</b>.</p>	<pre>replace(str,   'string', 'text') » text</pre> <pre>replace('string',   seq('s':'S',       't':'T')) » SString</pre>

Table 8: Search and replace functions and operators

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

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

#### 4.7.6 Comparing Strings

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

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

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

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

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

#### 4.7.7 Patterns and Captures

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

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

For performance reasons, you may use the following rule of thumb<sup>5</sup>:

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

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

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

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

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

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

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

**strings.match** extracts the hour.

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

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

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

---

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

> f():
1

> f():
10
```

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

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

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

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

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

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

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

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

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

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

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

Thus, since the string to be searched is longer,

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

returns:

```
null
```

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

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

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

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

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

> str:
value

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

> str:
value
```

Summary<sup>6</sup> of character classes and pattern modifiers:

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

Table 9: Character classes and modifiers

<sup>6</sup> Based on: `Programming in Lua`, 2nd edition, by Roberto Ierusalimsky, lua.org, pages 180f.

## 4.8 Boolean Expressions

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

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

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

Agena supports the following relational operators:

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

Table 10: Relational operators

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

Operator	Description	Examples
<b>and</b>	Returns its first operand if it is or evaluates to <b>false</b> , <b>fail</b> or <b>null</b> , otherwise returns its second operand.	$\text{true and } 1 \gg 1$ $\text{false and } 1 \gg \text{false}$ $\text{true and false} \gg \text{false}$ $\text{false and true} \gg \text{false}$
<b>or</b>	Returns its first operand if it is not or does not evaluate to <b>false</b> , <b>fail</b> , or <b>null</b> , otherwise it returns its second operand.	$\text{true or true} \gg \text{true}$ $\text{true or false} \gg \text{true}$ $2 \text{ or true} \gg 2$ $\text{null or } 2 \gg 2$
<b>xor</b>	With booleans: Returns the first operand if the second one evaluates or is <b>false</b> , <b>fail</b> , or <b>null</b> . It returns the second operand if the first operand evaluates to	$\text{true xor false} \gg \text{true}$ $\text{true xor true} \gg \text{false}$ $\text{false xor true} \gg \text{true}$ $1 \text{ xor null} \gg 1$ $1 \text{ xor } 2 \gg 2$

<sup>7</sup> See Chapter 4.9.3.



Operator	Description	Examples
	<b>false</b> , <b>fail</b> , or <b>null</b> and if the second operand is neither <b>false</b> , <b>fail</b> nor <b>null</b> .  With non-booleans: returns the first operand if the second operand evaluates to <b>null</b> , otherwise the second operand will be returned.	
<b>not</b>	Turns a true expression to <b>false</b> and vice versa. <b>not(fail)</b> become <b>true</b> .	<code>not true » false</code> <code>not false » true</code> <code>not 1 » false</code> <code>not null » true</code>
<b>nand</b>	Returns <b>true</b> if at least one operand is <b>false</b> , otherwise returns <b>false</b> .	<code>true nand false » true</code> <code>1 nand null » true</code>
<b>nor</b>	Returns <b>true</b> if both operands are <b>false</b> , and <b>false</b> otherwise.	<code>false nor false » true</code>
<b>xnor</b>	Returns <b>true</b> if both Boolean operands are the same (where <b>false</b> and <b>fail</b> are considered equal), and <b>false</b> otherwise.	<code>false xnor false » true</code>
<b>implies</b>	Returns <b>false</b> if the first operand is <b>true</b> and the second is <b>false</b> ; otherwise returns <b>true</b> .	<code>false implies false</code> <code>» true</code>

Table 11: Logical operators

As expected, you can assign Boolean expressions to names

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

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

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

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

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

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

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

```
> a := null

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

They are also handy to define defaults for unassigned names:

```
> a := null
> a := a or 0
> a:
0
```

You can add, subtract, multiply, divide and exponentiate numbers with **true** or **false**, where **true** in this context represents number 1 and **false** or **fail** number 0. Thus, for example, the expressions `abs(x > 0)*x` and `(x > 0)*x` are equivalent expressions. You can even apply the four basic arithmetic operations on two booleans if deemed necessary. See also **abs** and **signum** operators in Chapter 11.

## 4.9 Tables

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

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

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

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

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

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

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

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

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

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

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

#### 4.9.1 Arrays

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

`[ [ from k, ] value1 [, value2, ... ] ]`

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

The table *values* are 4, 5, and 6; by default, if no **from** clause is given, the numbers 1, 2, and 3 are the corresponding *keys* or *indices* of table *A*, with key 1 referencing value 4, key 2 referencing value 5, etc. With arrays, the indices always start with 1 and count upwards sequentially. The keys are always integral, so *A* in this example is an array whereas table *VL1* in the last chapter is not.

You can also pass an initialiser *k* for the first index. If given, the first value will have index *k*, the second value index *k* + 1 and so forth:

```
> A := [from 0, 4, 5, 6]:
[0 ~ 4, 1 ~ 5, 2 ~ 6]
```

(Technically, if *k* is not one, the result will most likely be a dictionary, see below, not an array.)

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

*tablename*[*key*]

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

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

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

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

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

```
tablename[key1][key2][...]
tablename[key1, key2, ...]
```

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

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

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

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

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

Tables may be nested:

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

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

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

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

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

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

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

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

```
> create table A;
```

```
> A.b:
null

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

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

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

```
> create table A;

> A[1]:
null

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

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

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

<i>tablename</i> [ <i>m to n</i> ]
------------------------------------

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

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

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

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

<b>create table</b> <i>name</i> <sub>1</sub> [, <b>table</b> <i>name</i> <sub>2</sub> , ...] <i>name</i> <sub>1</sub> := [ ]
---

```
> create table B;
```

creates the empty table B,

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

does exactly the same.

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

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

> B:
[a]
```

Alternatively, the **insert** statement always appends values to the end of a table<sup>8</sup>:

**insert** *value<sub>1</sub>* [, *value<sub>2</sub>*, ...] **into** *name*

```
> insert 'b' into B;

> B:
[a, b]
```

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

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

> B:
[2 ~ b]
```

The **delete**<sup>9</sup> statement works a little bit differently and removes all occurrences of a value from a table.

**delete** *value<sub>1</sub>* [, *value<sub>2</sub>*, ...] **from** *name*

```
> insert 'b' into B;

> delete 'b' from B;

> B:
[]
```

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

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

> delete 2 from B

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

---

<sup>8</sup> The **insert** statement cannot be applied on weak tables. See Chapter 6 for further information on this variant.

<sup>9</sup> ditto.

You can remove the holes in many cases, especially where order is not important, with functions **tables.hashole** combined with either **tables.entries** or **tables.reshuffle**, here is a code snippet:

```
> if tables.hashole(B) then tables.reshuffle(B, true) fi;

> B:
[1, 3]
```

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

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

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

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

```
> create table a(5);

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

## 4.9.2 Dictionaries

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

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

**[** [*key<sub>1</sub> ~ value<sub>1</sub>* [, *key<sub>2</sub> ~ value<sub>2</sub>*, ...]] **]**

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

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

Here is another example with strings as keys:

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

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

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

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

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

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

Further entries can be added with assignments such as:

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

which is the equivalent to

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

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

$$[ [name_1 = value_1 [, name_2 = value_2, \dots]] ]$$

Hence,

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

and

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

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

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

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

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



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

```
create dict name1(size1) [, dict name2(size2), ...]
```

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

```
create {table | dict} name1[(size1)] [, {table | dict} name2[(size2)], ...]
```

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

```
create table name1(arraysize1, hashsize1) [, ...]
```

#### 4.9.3 Table, Set and Sequence Operators

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

Name	Return	Function
<b>c in</b> A	Boolean	Checks whether the structure A contains the given value c.
<b>filled</b> A	Boolean	Determines whether a structure contains at least one value. If so, it returns <b>true</b> , else <b>false</b> .
<b>empty</b> A	Boolean	Checks whether a structure is empty.
A = B	Boolean	Checks whether two structures A, B contain the same values regardless of the number of their occurrence and order; if B is a reference to A, then the result is also <b>true</b> .
A <> B	Boolean	Checks whether two structures A, B do not contain the same values regardless of the number of their occurrence or order; if B is a reference to A, then the result is <b>false</b> .
A == B	Boolean	Checks whether two structures A, B contain the same number of elements and whether all key~value pairs in tables A, B or entries in the sets, sequences or registers are the same; if B is a reference to A, then the result is <b>true</b> .
<b>not</b> (A == B)	Boolean	The negation of A == B.
A ~= B	Boolean	Like ==, but checks the respective elements for approximate equality. Use <b>environ.kernel/eps</b> to change the setting for the accuracy threshold.

Name	Return	Function
<b>not</b> (A $\sim$ B)	Boolean	The negation of A $\sim$ B.
A <b>subset</b> B	Boolean	Checks whether the values in structure A are also values in B regardless of the number of their occurrence. The operator also returns <b>true</b> if A = B.
A <b>xsubset</b> B	Boolean	Checks whether the values in structure A are also values in B. Contrary to <b>subset</b> , the operator returns <b>false</b> if A = B.
A <b>union</b> B	table, set, seq, reg	Concatenates two tables, or two sets, or two sequences or registers A, B simply by copying all its elements - even if they occur multiple times - to a new structure. With sets, all items in the resulting set will be unique, i.e. they will not appear multiple times.
A <b>intersect</b> B	table, set, seq, reg	Returns all values in two structures A, B that are included both in A and in B and returns them in a new structure.
A <b>minus</b> B	table, set, seq, reg	Returns all the values in A that are not in B as a new structure.
<b>copy</b> A	table, set, seq, reg	Creates a deep copy of structure A, i.e. if A includes other tables, sets, pairs, sequences or registers, copies of these structures are created, too.
<b>join</b> A	string	Concatenates all strings in the table, sequence or register A.
<b>size</b> A	number	Returns the size of a table A, i.e. the actual number of key~value pairs in A. With sets, sequences and registers, the number of items will be returned.
<b>sort</b> (A)	table, seq, reg	This function sorts table, sequence or register A in ascending order. It directly operates on A, so it is destructive. With tables, the function has no effect on values that have non-integer keys. Note that <b>sort</b> is not an operator, so you must put the argument in brackets. Please also see Chapter 7 for its derivatives: <b>sorted</b> , <b>skycrane.sorted</b> , <b>stats.issorted</b> , and <b>stats.sorted</b> .
<b>unique</b> A	table, seq, reg	Removes multiple occurrences of the same value and returns the result in a new structure. With tables, also removes all holes (`missing keys`) by reshuffling its elements. This operator is not applicable to sets, since they are already unique.
<b>sumup</b> A	number	Sums up all numeric table, sequence or register values. If the structure is empty or contains no numeric values, <b>null</b> will be returned. Sets are not supported.
<b>qsumup</b> A	number	Raises each value in a table, sequence or register to the power of 2 and sums up these powers. If the structure is empty or contains no numeric values, <b>null</b> will be returned. Sets are not supported.

Name	Return	Function
$f @ A$	table, seq, set, reg	Maps a function $f$ on all elements of structure $A$ .
$f \$ A$	table, set, seq, reg	Selects all elements of a structure $A$ that satisfy a condition given by function $f$ .
$f \$\$ A$	Boolean	Checks whether at least one element in $A$ satisfies the condition checked by function $f$ .
$f \$\$\$ A$	number	Counts the number of elements in $A$ that satisfy the condition given by function $f$ .

Table 12: Table, set, and sequence operators

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

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

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

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

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

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

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

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

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

The **unique** function

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

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

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

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

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

returns **true**.

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

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

#### 4.9.4 Table Functions

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

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

Table 13: Basic table library procedures

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

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

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

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

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

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

Suppose we want to add a new entry 10 at position 3 of table  $c^{10}$ :

```
> C := [1, 2, 3, 4]
> put(C, 3, 10)
> C:
[1, 2, 10, 3, 4]
```

Now we remove this new entry 10 at position 3 again:

```
> purge(C, 3)
> C:
[1, 2, 3, 4]
```

Determine all elements in  $c$  that are even:

```
> select(<: x -> even(x) :>, C):
[2 ~ 2, 4 ~ 4]
```

Or return all elements not even:

```
> remove(<: x -> even(x) :>, C):
[1 ~ 1, 3 ~ 3]
```

Note that **remove** and **select** do not alter the original structure passed as the second argument. You can change this by passing the 'inplace' option which acts destructively:

```
> select(<: x -> even(x) :>, C, inplace = true):
[2 ~ 2, 4 ~ 4]
> C:
[2 ~ 2, 4 ~ 4]
```

---

<sup>10</sup> **put** and **purge** have to shift elements up or down, drawing performance. You may use the **llist** package to conduct these kinds of operations much faster in case of a large number of insertions or deletions.

**zip** zips together two tables by applying a function to each of its respective elements.

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

> zip(<: (x, y) -> x + y :>, C, [10, 20, 30, 40]):
[11, 22, 33, 44]
```

For other functions, have a look at Part II of this manual and the Agena Quick Reference Excel sheet.

#### 4.9.5 Table References

If you assign a table to a variable, only a reference to the table is stored in the variable. This means that if we have a table

```
> A := [1, 2];
```

assigning

```
> B := A;
```

does not copy the contents of A to B, but only the address of the same memory area which holds table [1, 2], hence:

```
> insert 3 into A;

> A:
[1, 2, 3]
```

also yields:

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

Use **copy** to create a true copy of the contents of a table. If the table contains other structures, copies of these structures are also made (so-called `deep copies`). Thus **copy** returns a new table without any reference to the original one.

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

> insert 4 into A;

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

With structures such as tables, sets, pairs, sequences or registers, all names to the left of an `->` token will point to the very same structure to its right.

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

```
> A[1] := 1

> B:
[1]
```

Tables can also directly or indirectly contain themselves, in which case they are also called `cycles`. Just some few examples:

```
> A := []

> A := [A, A]

> A:
[[], []]

> A.A := A

> A:
[1 ~ [], 2 ~ [], A ~ circum_table(0236A460)]
```

#### 4.9.6 Unpacking Tables by Name

There is syntactic sugar for the assignment statement to unpack named values, i.e. data indexed with string keys, from tables using the **in** keyword:

$$key_1 [, key_2, \dots] \text{ in } tablename$$

is equal to

$$key_1 [, key_2, \dots] := tablename.key_1 [, tablename.key_2, \dots]$$

A short example may suffice:

```
> zips := [duedo    = 40210:40629,
>          bonn     = 53111:53229,
>          cologne  = 50667:51149];

> duedo, bonn in zips

> duedo, bonn, cologne:
40210:40629      53111:53229      null
```

The **local** statement, see Chapter 6.2, supports this sugar, as well. Read also Chapter 5.2.12 for a variant implemented available in the **with** statement.

#### 4.9.7 Defining Multiple Constants Easily

The `// ... \` constructor allows to define a table of constant numbers and/or strings the simple way: items may not be separated by commas, and strings do not need to be put in quotes as long as they satisfy the criteria for valid variable names:

names starting with a hyphen or letter, including diacritics - and keywords such like **while**, **sqrt**, etc. do not have to be passed in quotes. Records are supported as well. Expressions like ``sin(0)`` etc. are *not* parsed and rejected. Example:

```
> a := // 0~0 1 2 3 zero one two three '2and3' sqrt ~ while \\:
[0 ~ 0, 1 ~ 1, 2 ~ 2, 3 ~ 3, 4 ~ zero, 5 ~ one, 6 ~ two, 7 ~ three,
8 ~ 2and3, sqrt ~ while]
```

As with table constructors, you can pass a start value for the first index:

```
> a := // from 0, 0 1 2 3 zero one two three '2and3' sqrt ~ while \\:
[0 ~ 0, 1 ~ 1, 2 ~ 2, 3 ~ 3, 4 ~ zero, 5 ~ one, 6 ~ two, 7 ~ three, 8 ~
2and3, sqrt ~ while]
```

## 4.10 Sets

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

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

Thus, they are equivalent to Cantor sets: An item is stored only once.

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

Besides being commonly used in mathematical applications, they are also useful to hold word lists where it only matters to see whether an element is part of a list or not:

```
> colours := {'red', 'green', 'blue'};
```

If you want to check whether the colour red is part of the set colours, just index it as follows:

`setname[item]`

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

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

If an item is not in the given set, the return is **false**. Note that we can use the same short form for indexing values (without quotes) as can be done with tables.

```
> colours.yellow:
false
```



If you want to add or delete items to or from a set, use the **insert** and **delete** statements. The standard assignment statement `setname[key] := value` is also supported.

**insert** *item<sub>1</sub>* [, *item<sub>2</sub>*, ...] **into** *name*

**delete** *item<sub>1</sub>* [, *item<sub>2</sub>*, ...] **from** *name*

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

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

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

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

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

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

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

When assigning items later, you will save at least 90 % of computation time if you know the size of the set in advance and initialise it with the maximum number of future entries as explained above. More items than stated at initialisation can be entered anytime, since Agena automatically enlarges the respective set accordingly and will also reserves space for additional entries.

Sets are useful in situations where the number of occurrence of a specific item or its position does not concern. Compared to tables, sets consume around 40 % less memory, and operations with them are 10 % to 33 % faster than the corresponding table operations.

Specifically, the more items you want to store, the faster operations will be compared to tables.

Note that if you assign a set to a variable, only a reference to the set is stored in the variable. Thus in a statement like `A := {}; B := A`, A and B point to the same set. Use the **copy** function if you want to create `independent` sets.

Sets can also include themselves, just an example:

```
> A := {}

> A := {A, A}:
{{}}
```

If you want to know the number of occurrence of a unique element in a distribution, the **bags** package might be of interest, see Chapter 10.8.

The following operators operate on sets:

Name	Return	Function
<b>c in A</b>	Boolean	Checks whether the set A contains the given value c.
<b>filled A</b>	Boolean	Determines whether a set contains at least one value. If so, it returns <b>true</b> , else <b>false</b> .
<b>empty A</b>	Boolean	Checks whether a set is empty.
<b>A = B</b>	Boolean	Checks whether two sets A, B contain the same values; if B is a reference to A, then the result is also <b>true</b> .
<b>A &lt;&gt; B</b>	Boolean	Checks whether two sets A, B do not contain the same values; if B is a reference to A, then the result is <b>false</b> .
<b>A == B</b>	Boolean	Same as =.
<b>A subset B</b>	Boolean	Checks whether the values in set A are also values in B. The operator also returns <b>true</b> if A = B.
<b>A xsubset B</b>	Boolean	Checks whether the values in set A are also values in B. Contrary to <b>subset</b> , the operator returns <b>false</b> if A = B.
<b>A union B</b>	set	Concatenates two sets A, B simply by copying all its elements to a new set. All items in the resulting set will be unique, i.e. they will not appear multiple times.
<b>A intersect B</b>	set	Returns all values in two sets A, B that are included both in A and in B as a new set.
<b>A minus B</b>	set	Returns all the values in A that are not in B as a new set.
<b>copy A</b>	set	Creates a deep copy of the set A, i.e. if A includes other tables, sets, pairs, sequences or registers, copies of these structures are built, too.
<b>size A</b>	number	Returns the size of a set A, i.e. the actual number of elements in A.
<b>f @ A</b>	set	Maps a function f on all elements of a set A.
<b>f \$ A</b>	set	Selects all elements in A that satisfy a given condition checked by function f.
<b>f \$\$ A</b>	Boolean	Checks the elements in A whether at least one satisfies a given condition checked by function f.

Table 14: Set operators

## 4.11 Sequences

Besides storing values in tables or sets, Agenda also features the sequence, an object which can hold any number of items except **null**. You may sequentially add items and delete items from it. Compared to tables, insertion and deletion are twice as fast with sequences. Contrary to all other data structures, Agenda automatically frees the memory occupied by a sequence if you remove values from it<sup>11</sup>.

Sequences store items in sequential order. As with tables, an item may be included multiple times. Sequences are usually indexed with positive integers in the same fashion as table arrays are, starting at index 1. If you pass a negative index  $n$ , then the  $|n|$ -th value from the right end, i.e. the top of the sequence will be determined. Non-integral indices are not allowed. As with tables, you can compute the index in assignments or queries.

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

$$\mathbf{seq}([item_1 [, item_2, \dots]])$$

```
> a := seq(0, 1, 2, 3);
```

```
> a:
seq(0, 1, 2, 3)
```

You can access the items the usual way:

$$seqname[index]$$

```
> a[1]:
0
```

```
> a[2], a[3]:
1    2
```

If the index is larger than the current size of the sequence, an error will be returned<sup>12</sup>.

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

Sublists of sequences can be determined with the following syntax:

$$seqname[m \text{ to } n]$$

---

<sup>11</sup>You can turn off this feature by issuing: `environ.kernel(seqautoshrink = false)`.

<sup>12</sup> The error message can be avoided by defining an appropriate metamethod.

Agena returns all values from and including index position  $m$  to  $n$ , with  $m$  and  $n$  positive or negative integers. In case of a non-existing key, an error will be issued. If  $m > n$ , an empty sequence will be returned.

```
> a[2 to 3]:
seq(1, 2)
```

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

In general, the **settype** function allows you to set a user-defined subtype for a sequence, set, table or pair.

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

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

```
> gettype(a):
duo
```

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

Once the type of a sequence has been set, the **typeof** operator also returns this user-defined sequence type and will not return 'sequence'.

```
> typeof(a), gettype(a):
duo    duo
```

This allows you to programme special operations only applicable to certain types of sequences.

The **::** and **-:** operators can check user-defined types. Just pass the name of your type as a string:

```
> a :: 'duo':
true
> a -: 'duo':
false
```

Note that if a user defined-type has been given, the check for a basic type with the **::** and **-:** operators will return also return **true** or **false**.

```
> a :: sequence:
true
> a -: sequence:
false
```

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

```
> settype(a, null);

> typeof(a):
sequence
```

The **create sequence** statement creates an empty sequence and optionally allows to allocate enough memory in advance to hold a given number of elements (which can be inserted later). Agena automatically will extend the sequence, if the predetermined number of items is exceeded. The **sequence** and **seq** keywords are synonyms.

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

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

```
> create sequence a(4);

> insert 1 into a;

> a[2] := 2;

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

Note that if the index is larger than the number of items stored to it plus 1, Agena returns an error in assignment statements, since `holes` in a sequence are not allowed. The next free position in *a* is at index 3, however a larger index is chosen in the next example.

```
> a[4] := 4
Error, line 1: index out of range

> a[3] := 3
```

Items can be deleted by setting their index position to **null**, or by applying **delete**, i.e. stating which items - not index positions - shall be removed. Note that all items to the right of the value deleted are shifted to the left, thus their indices will change.

```
> a[1] := null

> a:
seq(2, 3)

> delete 2, 3 from a

> a:
seq()
```

Thus concerning the **insert** and **delete** statements, we have the following familiar syntax:

**insert**  $item_1$  [,  $item_2$ , ...] **into** *name*

**delete**  $item_1$  [,  $item_2$ , ...] **from** *name*

If you assign a sequence to a variable, only a reference to the sequence is stored in the variable. Thus sequences behave the same way as tables and sets do, i.e. in a statement like `A := seq(); B := A`, A and B point to the same sequence in memory. Use the **copy** function if you want to create `independent` sequences.

```
> A := seq()
> B := A
> A[1] := 10
> B:
seq(10)
```

As with tables and sets, sequences can also reference to themselves:

```
> A := seq()
> A[1] := A
> A[2] := A
> A:
seq(circum_sequence(01E647D8), circum_sequence(01E647D8))
```

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

Name	Description	Example
=	Equality check the Cantor way	<code>a = b</code>
==	Strict equality check	<code>a == b</code>
~=	approximate equality check	<code>a ~= b</code>
<>	Inequality check the Cantor way	<code>a &lt;&gt; b</code>
::	Type check operator	<code>a :: sequence</code> <code>a :: 'usertype'</code>
:-	Negation of type check operation	<code>a -: sequence</code> <code>a -: 'usertype'</code>
@	Maps a function on all elements of a sequence.	<code>f @ a</code>
\$	Selects all elements of A that satisfy a given condition.	
\$\$	Checks whether at least one element in A satisfies a condition.	<code>f \$\$ a</code>
<b>insert</b>	Inserts one or more elements.	<code>insert 1 into a</code>
<b>delete</b>	Deletes one or more elements.	<code>delete 0, 1</code> <code>from a</code>

Name	Description	Example
<b>bottom</b>	Returns the item with key 1.	bottom a
<b>top</b>	Returns the item with the largest key.	top a
<b>pop</b>	as an operator works like top but also removes the item from the sequence	pop a
<b>copy</b>	Creates an exact copy of a sequence; deep copying is supported so that structures inside sequences are properly treated.	copy a
<b>filled</b>	Checks whether a sequence has at least one item.	filled a
<b>empty</b>	Checks whether a sequence is empty.	empty a
<b>getentry</b>	Returns entries without issuing an error if a given index does not exist.	getentry(a, 1, 3)
<b>in</b>	Checks whether an element is stored in the sequence, and returns <b>true</b> or <b>false</b> . See also <b>binsearch</b> .	0 in seq(1, 0)
<b>join</b>	Concatenates all strings in a sequence in sequential order.	join(a)
<b>pop</b>	Pops the first or the last element from a sequence.	pop bottom from a pop top from a
<b>size</b>	Returns the current number of items.	size a
<b>sort</b>	Sorts a sequence in place. Please also see Chapter 7 for its derivatives: <b>sorted</b> , <b>skycrane.sorted</b> , <b>stats.issorted</b> , and <b>stats.sorted</b> .	sort(a)
<b>purge(o, i)</b>	Deletes the value o[i]	purge(a, 2)
<b>subs(x:v, o)</b>	Substitutes all occurrences of value x in o with value v.	subs(8:0, seq(1, 8))
<b>subsop(i:v, o)</b>	Substitutes the value at index i in o with value v.	subsop(1:0, seq(1, 2))
<b>type</b>	Returns the general type of a sequence, i.e. <b>sequence</b> .	type a
<b>typeof</b>	Returns the user-defined type of a sequence, or the basic type if no special type has been defined.	typeof a
<b>unique</b>	Reduces multiple occurrences of an item in a sequence to just one.	unique a
<b>unpack</b>	Unpacks a sequence. See <b>unpack</b> in Chapter 8.	unpack(a)
<b>nseq</b>	Creates a new sequence and fills it with values	nseq(<: x -> x :>, 1, 10)
<b>map</b>	Maps a function on all elements of a sequence.	map(<: x -> x^2 :>, seq(1, 2, 3))
<b>zip</b>	Zips together two sequences by applying a function to each of its respective elements.	zip(<: x, y -> x + y :>, seq(1, 2), seq(3, 4))

Name	Description	Example
<b>intersect</b>	Searches all values in one sequence that are also values in the other sequence and returns them in a new sequence.	<code>seq(1, 2)</code> <code>intersect</code> <code>seq(2, 3)</code>
<b>minus</b>	Searches all values in one sequence that are not values in the other sequence and returns them as a new sequence.	<code>seq(1, 2)</code> <code>minus seq(2, 3)</code>
<b>subset</b>	Checks whether all values in a sequence are included in the other sequence.	<code>seq(1)</code> <code>subset seq(1, 2)</code>
<b>union</b>	Concatenates two sequences simply by copying all its elements.	<code>seq(1, 2)</code> <code>union seq(2, 3)</code>
<b>settype</b>	Sets a user-defined type for a sequence.	<code>settype(a, 'duo')</code>
<b>gettype</b>	Returns a user-defined type for a sequence.	<code>gettype(a)</code>
<b>setmetatable</b>	Assigns a metatable to a sequence.	<code>setmetatable</code> <code>(a, mtbl)</code>
<b>getmetatable</b>	Returns the metatable stored to a sequence.	<code>getmetatable(a)</code>

Table 15: Basic sequence operators and functions

For more functions, consult the Agena Quick Reference Excel sheet. Also, you may have a look at the **llist** linked list package presented in Chapter 6.28, if you have to conduct a lot of insertions and/or deletions in a data structure.

The `(/ ... \)` constructor allows to define a sequence of constant numbers and/or strings the simple way: items may not be separated by commas, and strings do not need to be put in quotes as long as they satisfy the criteria for valid variable names (name starting with a hyphen or letter, including diacritics) or if they are keywords. Expressions like ``sin(0)`` etc. are rejected. Example:

```
> a := (/ 0 1 2 3 zero one two three '2and3' while \):
seq(0, 1, 2, 3, zero, one, two, three, 2and3), while]
```

## 4.12 Stack Programming

Sequences and sometimes table arrays can be used to implement stacks, and besides the **insert/into** statement to put an element to the top, an efficient statement is available to remove an item from the bottom or from the top of the stack:

**pop bottom from** *name*

**pop top from** *name*

Both variants work on tables even if their integer keys are not distributed consecutively.



The **bottom** and **top** operators return the element at the bottom of the stack and the top of the stack, respectively. They both do not delete the element returned from the stack.

```
> stack := seq();  
  
> insert 10, 11, 12 into stack;  
  
> bottom(stack):  
10  
  
> top(stack):  
12  
  
> pop bottom from stack;  
  
> pop top from stack;  
  
> stack:  
seq(11)
```

The **rotate** statement moves each element in a sequence, register or the array part of a table one position to the bottom (downwards) or to the top (upwards):

<p><b>rotate bottom</b> <i>name</i></p> <p><b>rotate top</b> <i>name</i></p>
--

The element at the bottom or the top is moved to the top or the bottom, respectively. See also the **shift** function described in Chapter 8 of the Reference.

```
> s := seq(1, 2, 3);  
  
> rotate bottom s;  
  
> s:  
seq(2, 3, 1)  
  
> s := seq(1, 2, 3):  
seq(1, 2, 3)  
  
> rotate top s;  
  
> s:  
seq(3, 1, 2)
```

The **pop** operator - contrary to **top** - both returns the top element of a sequence or register and then removes it from the structure. With tables, it returns the value indexed by the largest integer key and then also removes it.

```
> pop(s):  
2  
  
> s:  
seq(3, 1)
```

There are two other statements that work on sequences and registers only: The **exchange** statement swaps the two topmost elements, and the **duplicate** statement adds a copy of the current topmost element to the end of the structure.

```
> exchange s

> s:
seq(1, 3)

> duplicate s

> s:
seq(1, 3, 3)
```

You may try to use the **put** function to insert new values in the interior of a stack, shifting up other values to open space, and **purge** to delete values in the interior of a stack.

See also Chapter 14.6 of the Reference for the six built-in number and character stacks. You might also check out Chapter 10.15 and 10.16 for first-in first-out and last-in first-out queues.

### 4.13 More on the create Statement

You cannot only initialise any table arrays with the **create** statement, but also dictionaries, sets, and sequences with only one call and in random order, so the following statement is valid:

```
> create table a, dict b(10), set c, sequence d(100), table e(10);

> a, b, c, d, e:
[]      []      {}      seq()    []
```

### 4.14 Pairs

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

$item_1 : item_2$
-------------------

```
> p := 1:2

> p:
1:2
```

The **left** and **right** operators provide read access to its left and right operands; the standard indexing method using indexed names is supported, as well:

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

```
> left(p), p[1]:
1      1

> right p, p[2]:
2      2
```

An operand of an existing pair can be changed by assigning a new value to an indexed name, where the left operand is indexed with number 1, and the right operand with number 2:

```
> p[1] := 2;

> p[2] := 3;
```

You can compute the index as long as the result evaluates to the integers 1 or 2, as well.

As with sequences, you may define user-defined types for pairs with the **settype** function which also changes the way pairs are output.

```
> typeof(p):
pair

> settype(p, 'duo');

> p:
duo(2, 3)

> typeof(p):
duo

> gettype(p):
duo

> p :: pair:
true

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

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

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

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

```
> 2 in 0:10:
true

> 's' in 0:10:
fail
```

As with all other structures, if you assign a pair to a variable, only a reference to the pair is stored in the variable. Thus in a statement like `A := a:b; B := A`, A and B point to the same pair. Use the **copy** function if you want to create ‘independent’ pairs.

Summary:

Name	Description	Example
<code>=, ==, ~=</code>	Equality checks (mostly same functionality)	<code>a = b</code>
<code>&lt;&gt;</code>	Inequality check	<code>a &lt;&gt; b</code>
<code>::</code>	Type check operator	<code>a :: pair</code> <code>a :: 'udeftype'</code>
<code>-:</code>	Negation of type check operation	<code>a -: pair</code> <code>a -: 'udeftype'</code>
<code>@</code>	Maps a function on each operand.	<code>f @ a</code>
<b>copy</b>	Creates an exact copy of a pair; deep copying is supported so that structures inside pairs are properly treated.	<code>copy a</code>
<b>in</b>	If the left operand <code>x</code> is a number and if the left and right hand side of the pair <code>a:b</code> are numbers, then the operator checks whether <code>x</code> lies in the closed interval <code>[a, b]</code> and returns <b>true</b> or <b>false</b> . If at least one value <code>x</code> , <code>a</code> , <code>b</code> is not a number, the operator returns <b>fail</b> .	<code>1.5 in 1:2</code>
<b>left</b>	Returns the left operand of a pair.	<code>left(a)</code>
<b>right</b>	Returns the right operand of a pair.	<code>right(a)</code>
<b>type</b>	With pairs, always returns <code>'pair'</code> .	<code>type(a)</code>
<b>typeof</b>	Returns either the user-defined type of the pair, or the basic type ( <code>'pair'</code> ) if no special type was defined for the pair.	<code>typeof(a)</code>
<b>settype</b>	Sets a user-defined type for a pair.	<code>settype(a, 'duo')</code>
<b>gettype</b>	Returns the user-defined type of a pair.	<code>gettype(a)</code>
<b>setmetatable</b>	Sets a metatable to a pair.	<code>setmetatable(p, mtbl)</code>
<b>getmetatable</b>	Returns the metatable stored to a pair.	<code>getmetatable(p)</code>

Table 16: Operators and functions applicable to pairs

## 4.15 Registers

Registers are memory-efficient, fixed-size Agena `sequences` that also store **null**'s. They are not automatically extended if more values have to be added, but can be manually resized.

Registers allow to hide data: by changing the pointer to the top of a register using **registers.settop**, any values stored above (the position of) this pointer can neither be read nor changed by any of Agena's functions and operators. Registers are supported by most of the existing statements, operators and functions. Please also refer to Chapter 6.15 `Sandboxes`.

The concept of the fixed size and the top pointer is key to understanding and working with registers.

By default, the top pointer always refers to the very last element in a register - it is automatically changed only if an element is removed with the **pop top** or **pop bottom** statements, the **pop** operator or the **purge** function.

In general, registers can save memory if you know the precise number of values to be stored, or to be added or removed later, in advance. As such, they behave like C arrays storing any value without provoking faults. With respect to sequences, there usually are no performance gains with most operations - but since registers do not automatically shift elements, they are eight times faster when deleting items.

Let us first create a register with eight items:

```
> a := reg(1, 2, 3, 4, 5, 6, 7, 8):
reg(1, 2, 3, 4, 5, 6, 7, 8)
```

Read the first element:

```
> a[1]:
1
```

Set the first entry to **null** - contrary to other data structures, the size of register is not reduced, and no values are shifted.

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

> a:
reg(null, 2, 3, 4, 5, 6, 7, 8)
```

Now reset the pointer to the top of the register to the fourth element:

```
> registers.settop(a, 4);

> size(a):
4

> a:
reg(null, 2, 3, 4)
```

```
> a[5]:
In stdin at line 1:
  Error: register index 5 out of current range.

Stack traceback:
  stdin, at line 1 in main chunk
```

By changing the position of the top pointer beyond 4, we can read and change the values again:

```
> registers.settop(a, 8);

reg(null, 2, 3, 4, 5, 6, 7, 8)
```

When passing no elements to the **reg** operator, by default a register with sixteen slots is created.

```
> reg():
reg(null, null, null, null, null, null, null, null, null, null, null, null,
null, null, null, null)
```

But you can change this default to another value:

```
> environ.kernel(regsize = 8);

> reg():
reg(null, null, null, null, null, null, null, null)
```

Registers containing **null**'s may issue errors with some functions or operators.

Changing the **size** of a register at runtime is easy:

```
> b := reg('a', 'b', 'c'):
reg(a, b, c)
```

**register.resize** enlarges a register to the given number of elements.

```
> registers.resize(b, 8);

> b:
reg(a, b, c, null, null, null, null, null)
```

**register.resize** can also shrink a register to the given number of elements.

```
> registers.resize(b, 4);

> b:
reg(a, b, c, null)
```

Registers support metamehtods and user-defined types. To hide the current size of the register as defined above, we could assign:

```
> size a:
8
```

```
> mt := [
>   '___size' ~ proc(x) is
>     return 0
>   end
> ]

> setmetatable(a, mt);

> size a:
0
```

Last but not least, the **create register** statement creates an empty register and optionally allows to allocate enough memory in advance to hold a given number of elements which can be inserted later. The **register** and **reg** keywords are synonyms.

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

By default, if no number is given in brackets, the statement creates a register with 16 slots pre-filled with **nulls**. If a positive integer is given in brackets, a register with exactly this number of slots is created.

Items can be inserted with the **insert** statement, replacing each **null** value that may exist, and from the start of the structure. You may also use the conventional indexing method.

```
> create reg r(4)

> r[2] := 20

> r:
reg(null, 20, null, null)

> insert 10 into r

> r:
reg(10, 20, null, null)

> insert 30 into r

> r:
reg(10, 20, 30, null)
```

Name	Description	Example
=	Equality check the Cantor way	a = b
==	Strict equality check	a == b
~=	Approximate equality check	a ~= b
<>	Inequality check the Cantor way	a <> b
::	Type check operator	a :: register
:-	Negation of type check operation	a -: register
@	Maps a function f on all elements of register a.	f @ a

Name	Description	Example
\$	Selects all elements in a of a that satisfy a condition given by function f.	f \$ a
\$\$	Checks whether at least one element in a satisfies a condition given by function f.	f \$\$ a
>>	Counts the number of items in a that satisfy the condition given by function f.	f >> a
insert	Inserts an element at the first position that holds a <b>null</b> value.	insert 0, 1 into a
delete	Deletes one or more elements and replaces them with <b>null</b> .	delete 0, 1 from a
bottom	Returns the item with key 1.	bottom a
top	Returns the item with the largest key.	top a
pop	as an operator works like <b>top</b> but also removes the item from the register.	pop a
copy	Creates an exact copy of a register; deep copying is supported so that structures inside register are properly treated.	copy a
filled	Checks whether a register has at least one item, including null. This is always true.	filled a
getentry	Returns entries without issuing an error if a given index does not exist.	getentry(a, 1, 3)
in	Checks whether an element is stored in the register, returns <b>true</b> or <b>false</b> .	0 in reg(1, 0)
pop bottom/ top	Pops the first or the last element from a register, shifting other elements to close the space, if necessary. Reduces the size of the register by one.	pop bottom from a pop top from a
size	Returns the number of `visible` elements.	size a
sort	Sorts a register in place. Please also see <b>sorted</b> .	sort(a)
type	Returns the general type of a register, i.e. register.	type a
unique	Reduces multiple occurrences of an item in a register to just one.	unique a
unpack	Unpacks a register. See <b>unpack</b> in Chapter 8.	unpack(a)
duplicates	Finds duplicate elements.	duplicates(a)
map	Maps a function on all elements of a register.	map(<: x -> x^2 :>, reg(1, 2, 3))
purge	Removes the value at the given position and shifts all elements to close the space. Also reduces the size of the register by one.	purge(a, 2)
subs(x:v, o)	Substitutes all occurrences of value x in o with value v.	subs(8:0, reg(1, 8))
subsop(i:v, o)	Substitutes the value at index i in o with value v.	subsop(1:0, reg(1, 2))



Name	Description	Example
<b>zip</b>	Zips together two registers by applying a function to each of its respective elements.	<code>zip(&lt;: x, y -&gt; x + y :&gt;, reg(1, 2), reg(3, 4))</code>
<b>intersect</b>	Searches all values in one register that are also values in another register and returns them in a new register.	<code>reg(1, 2) intersect reg(2, 3)</code>
<b>minus</b>	Searches all values in one register that are not values in another register and returns them as a new register.	<code>reg(1, 2) minus reg(2, 3)</code>
<b>subset</b>	Checks whether all values in a register are included in another register.	<code>reg(1) subset reg(1, 2)</code>
<b>xsubset</b>	Checks whether all values in a register are included in another register.	<code>reg(1) xsubset reg(1, 2)</code>
<b>union</b>	Concatenates two registers simply by copying all its elements.	<code>reg(1, 2) union reg(2, 3)</code>
<b>setmeta-table</b>	Assigns a metatable to a register.	<code>setmetatable (a, mtbl)</code>
<b>getmeta-table</b>	Returns the metatable stored to a register.	<code>getmetatable(a)</code>
<b>registers. settop</b>	Resets the top pointer to the given position, an integer.	
<b>registers. resize</b>	Shrinks or extends the size of a register to the given number of slots.	
<b>environ. kernel/ regsize</b>	Sets the default size of newly created registers the given value, a non-posint.	

Table 17: Some operators and functions applicable to registers

### 4.16 Exploring the Internals of Structures

If you would like to know how a table, set, sequence, register or pair is represented internally, please have a look at the **environ.attrib** function explained in Chapter 14.2. It might help when debugging code.

The function returns the estimated number of bytes used by a structure, how many slots have been pre-allocated and how many are actually occupied, whether a user-defined type has been set, how many elements have been allocated to the array and hash parts of a table, etc.

### 4.17 Other Types

For threads, userdata, and lightuserdata please refer to the Lua 5.1 documentation and Chapter 6.30.

Agena supports the following metamethods with all data types: **=**, **==**, **~=**, **size**, **in**, **union**, **intersect**, **minus**, **mulup**, **sumup**, **qsumup** and **qmdev**. **'\_\_index'** , **'\_\_writeindex'**, **'\_\_gc'**, and **'\_\_tostring'** are supported, as well.

## Chapter **Five**

# Control



## 5 Control

### 5.1 Conditions

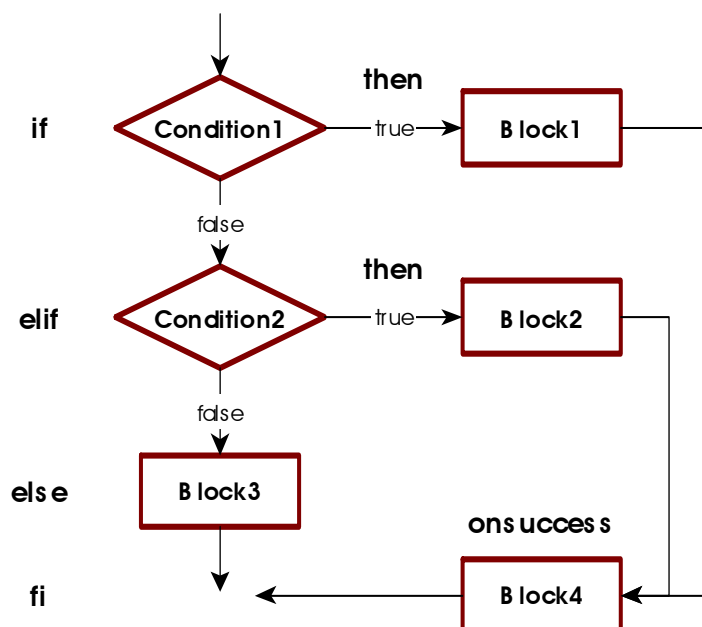
Depending on a given condition, Agenda can alternatively execute certain statements with either the **if** or **case** statement.

#### 5.1.1 if Statement

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

```
if condition1 then
  statements1
[elif condition2 then
  statements2]
[onsuccess
  statements3]
[else
  statements4]
fi
```

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



The **elif**, **else**, and **onsuccess** clauses are optional. While more than one **elif** clause can be given, only one **else** and one **onsuccess** clause is accepted.

If an **if** or **elif** condition results to **true** or any other value except **false**, **fail** or **null**, its corresponding **then** clause is executed. If all conditions result to **false**, **fail** or **null**, the **else** clause is executed if present - otherwise Agenda proceeds with the next statement following the **fi** keyword.

If an **onsuccess** clause is given, and an **if** or **elif** condition results to **true**, the statements in this **onsuccess** branch are executed. This allows to move code common to all **then** clauses into one single branch, reducing code size. When

using both **onsuccess** and **else** clauses, the **onsuccess** clause must be put given before the else snippet.

Examples:

The condition **true** is always true, so the string 'yes' is printed.

```
> if true then
>   print('yes')
> fi;
yes
```

The next example demonstrates the behaviour if the condition is neither a Boolean nor **null**:

```
> if 1 then
>   print('One')
> fi;
One
```

In the following statement, the condition evaluates to **false**, so nothing is printed:

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

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

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

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

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

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

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

Sometimes certain conditions may just be skipped with an empty statement, denoted by **do nothing**, to make the code more readable:

```
> if 1 = 2 then
>   do nothing
> elif 1 < 2 then
>   print('this will always be printed')
> else
>   print('neither will this be printed')
> fi;
this will always be printed
```

One last example, this time demonstrating the optional **onsuccess** clause. As shown, both **then** statements include the same `flag := true` statement.

```
> if 1 = 2 then
>   print('this will never be printed');
>   flag := true
> elif 1 = 1 then
>   print('this will always be printed');
>   flag := true
> else
>   flag := false
> fi;
this will always be printed

> flag:
true
```

So the two assignment statements may be moved into one **onsuccess** clause.

```
> if 1 = 2 then
>   print('this will never be printed');
> elif 1 = 1 then
>   print('this will always be printed');
>   onsuccess
>     flag := true
> else
>   flag := false
> fi;
this will always be printed

> flag:
true
```

**if** and **elif** statements also support simple assignments in the conditions, as well.

```
> if flag := true then
>   print('Output: ' & flag)
> fi;
Output: true
```

Only if the right-hand side of the assignment does neither result to **false**, **fail** nor **null**, will the corresponding **then** clause be executed.

You can also combine an assignment and a condition in the **if** clause:

```
> if c := 0, c >= 0 do
>   print(c)
> od;
0
```

You can combine local declarations and assignments in **if** statements. So instead of:

```
> scope
>   local c;
>   if c := 0, c >= 0 do
>     print(c)
>   od;
>   print(c)
> epocs
0
0
```

the following will do the same:

```
> scope
>   if local c := 0, c >= 0 do
>     print(c)
>   od;
>   print(c)
> epocs
0
0
```

Note that the locally declared variable is not only valid in the **then/fi** block, but also in the surrounding block.

### 5.1.2 if Operator, Version One

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

[with name<sub>1</sub>, ... := expr<sub>1</sub>, ... [->]]  
**if** condition<sub>1</sub> **then** expr<sub>1</sub> [**elif** condition<sub>2</sub> **then** expr<sub>2</sub>, ...] **else** expr<sub>k</sub> **fi**

The result is expression expr<sub>1</sub> if condition<sub>1</sub> is **true** or any other value except **false**, **fail** or **null**; and expr<sub>k</sub> otherwise. You can also optionally add one or more **elif** clauses.

Example:

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

which is the same as:



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

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

An optional preceding **with** clause allows to define one or more auxiliary variables that are local to this operator only:

```
> x := Pi;

> a := with n := 2*x -> if x < 0 then n else 2*n fi;
```

which is syntactic sugar for:

```
> x := Pi;

> scope
>   local n := 2*x;
>   a := if x < 0 then n else 2*n fi
> epocs;
```

The arrow token is optional. Multiple auxiliary variables are defined as follows:

```
> a := with m, n := x, 2*x -> if x < 0 then m else n fi;
```

The **if** operator cannot return multiple values, only one.

### 5.1.3 if Operator, Version Two

There is a second operator form, reminiscent to the **if** statement; for example:

```
> a := 10;

> sgn := if is a < 0 then # determines sign of `a'
>         print('I am negative');
>         [further statements ...]
>         return -1
>     elif a = 0 then
>         print('I am zero');
>         return 0
>     else
>         return 1
>     fi;
> sgn:
1
```

You may omit the **elif** and **else** clauses. Each clause may contain zero, one or more statements, but it must always finish with the **return** expression which defines the resulting value (-1, 0 or 1 in the example above). In procedures, this special **return** expression does not cause a procedure to quit. Note that if the **else** clause is omitted, the operator returns **null** if no condition is met.

The operator returns exactly one value.

#### 5.1.4 Short-cut Condition with ? and -? Tokens

The question mark **?** expresses a shortcut `if`-like statement: if any condition preceding **?** evaluates to true, exactly one statement right after the token is executed, otherwise the statement is simply skipped. Likewise, the **-?** token checks an expression and executes a one-line statement if it evaluates to **false**, **fail** or **null**.

```
> x := 0;

> x = 0 ? x := 1;

> x:
1

> x := 0;

> x <> 0 -? x := 1;

> x:
1
```

### 5.1.5 case Statement

The **case** statement facilitates comparing values and executing corresponding statements. There exist two variants, the first one is:

```

case name
  [of value11 [, value12, ...] then statements1
  [of value21 to value22 then statements2]
  [of ...]
  [onsuccess ...]
  [else statementsk [esle]]
esac

```

```

> a := 'k';

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

> result:
consonant

```

You can add as many **of/then** statements as you like. Fall through is not supported. This means that if one **then** clause is executed, Agena will not evaluate the following **of** clauses and will proceed with the statement right after the closing **esac** keyword. An **else** clause may be terminated by the **esle** token, but this is optional.

Instead of passing one or more individual values, you can also check whether a number  $x$  or the first character of a - non-empty - string  $x$  is part of a range  $a$  to  $b$ , i.e.  $a \leq x \leq b$ . One **to** range is accepted per **of** clause.

```

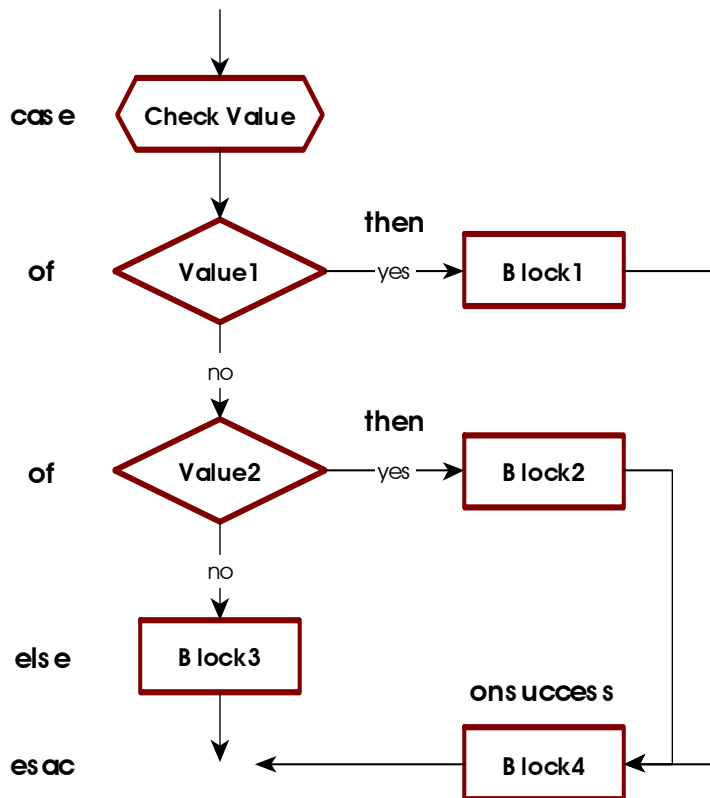
> a := 0;

> case a
>   of -1 then result := -1
>   of 0 to 10 then result := 10
>   of 'a' to 'c' then result := 0
> esac;

```

As with the **if** statement, if an **onsuccess** clause is given, and in case one of the conditions results to **true**, the statements in the **onsuccess** branch are executed. This allows to move code common to all **then** clauses into one single branch, reducing the code size.

If none of the **of** conditions is satisfied, and if an **else** clause is given, then the respective **else** statements will be processed, otherwise Agena executes the code following the **esac** token.



The second variant is exactly equal to the **if** statement but may improve the readability of programme code.

With both variants, instead of the **then** keyword the **->** token can be used.

### 5.1.6 case of Statement

A flavour of the **if** statement is the **case of** control. It may improve the readability of code.

There is no functional difference between **if** and **case of** statements.

Example:

```

> x := 0; flag := false;

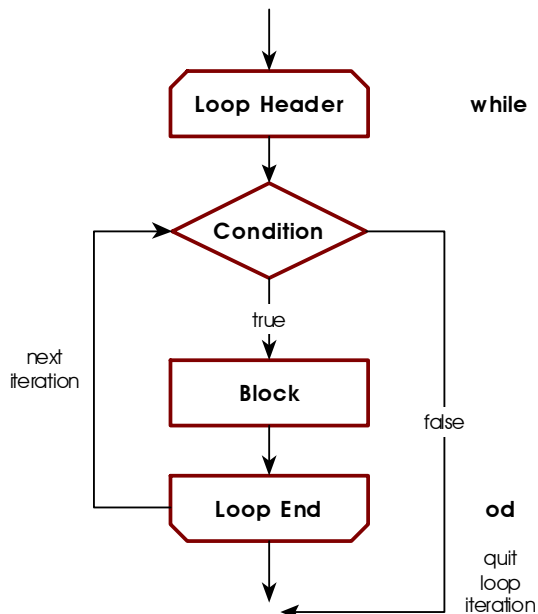
> case
>   of x < 0 then r := -1;
>   of x = 0 then r := 0;
>   onsuccess flag := true;
>   else r := 1 esle
> esac

> r, flag:
0   true
  
```

```

case
  of condition1 then statements1
  [of condition2 then statements2]
  [of ...]
  [onsuccess ...]
  [else statementsk [esle]]
esac
  
```

**case of** statements also support simple assignments in the **case of** clause, and their optional **of** clauses, as well.



```

> case of flag := io.read() then
>   print('Output: ' & flag)
> esac;
Agena
Output: Agena

```

Only if the right-hand side of the assignment does neither result to **false**, **fail** nor **null** will the **then** clause be executed.

## 5.2 Loops

Agena has three basic forms of control-flow statements that perform looping: **while** and **for**, each with different variations.

### 5.2.1 while Loops

A **while** loop first checks a condition and if this condition is **true** or any other value except **false**, **fail** or **null**, it iterates the loop body again and again as long as the condition remains true.

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

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

```

while condition do
  statements
od

```

The programme flow is as shown in the diagram above.

The following statements calculate the largest Fibonacci number less than 1000.

```

> a := 0; b := 1;

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

> c:
987

```

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

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

You can also conduct a simple assignment in the **while** condition. If an assignment is given in the **while** clause, its right-hand side is evaluated and stored to the left-hand side name. The result of the evaluation is then checked and either the loop body is executed - the result of the evaluation is neither **false**, **fail** nor **null** - or not.

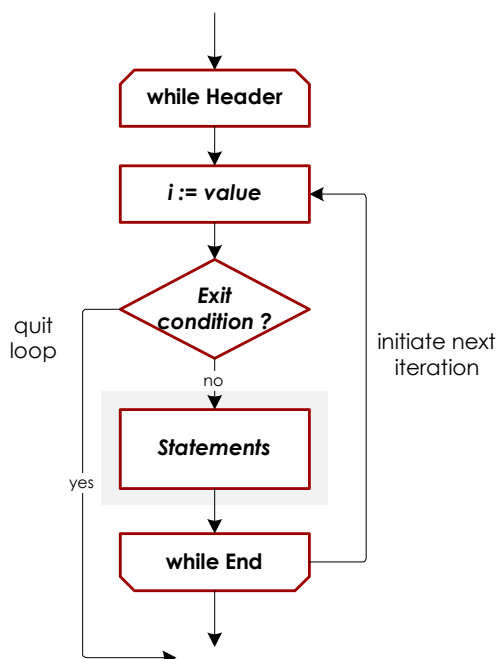
This allows for shorter code: Instead of

```
> flag := true;
> while flag do
>   flag := io.read();
>   if flag = 'Z' then break fi
> od
```

with no need to assign `flag` before, you can now simply write:

```
> while flag := io.read() do
>   if flag = 'Z' then break fi
> od
```

The variable assigned in the **while** clause is *not* local to the loop body but can be accessed later on the level that is surrounding the loop. You may explicitly declare the variable locally before, but this is not obligatory.



You can combine an assignment and a condition in the **while** clause. In this case, *the assignment statement in the while clause is always redone* when the control flow is returning to the start of the **while** loop, before deciding whether to execute the loop body once again.

```
> i := 0.1;
> while logn := ln(i), logn < -0.9 do
>   print(i, logn);
>   i += 0.1
> od;
0.1      -2.302585092994
0.2      -1.6094379124341
0.3      -1.2039728043259
0.4      -0.91629073187416
```

You can combine local declarations and assignments in **while** statements. For example, with the previous example:

```
> i := 0.1;

> while local logn := ln(i), logn < -0.9 do
>   print('>', i, logn);
>   i += 0.1
> od;
```

Note that the locally declared variable is not only valid in the **do/od** block, but also in the surrounding block.

Variations of **while** are the **do/as** and **do/until** loops which check a condition at the end of the iteration, and thus will always be executed at least once.

In the **do/as** variant, as long as the condition evaluates to **true**, the loop body is executed.

```
> c := 0;

> do
>   inc c
> as c < 10;

> c:
10
```

```
do
  statements
as condition
```

**do/until** loops are iterated until the given condition is met.

```
> c := 0;

> do
>   inc c
> until c > 10;

> c:
11
```

```
do
  statements
until condition
```

**do/as** and **do/until** support simple assignments in the respective condition.

Another flavour of the **while** loop is the infinite **do/od** loop which executes statements infinitely and can be interrupted with the **break** or **return** statements. See Chapter 5.2.10 for further information on the **break** statement. It is syntactic sugar for the **while true do/od** construct.

```
do
  statements
od
```

```
> i := 0;  
> do  
>   inc i;  
>   if i > 3 then break fi;  
>   print(i)  
> od;
```

```
1  
2  
3
```

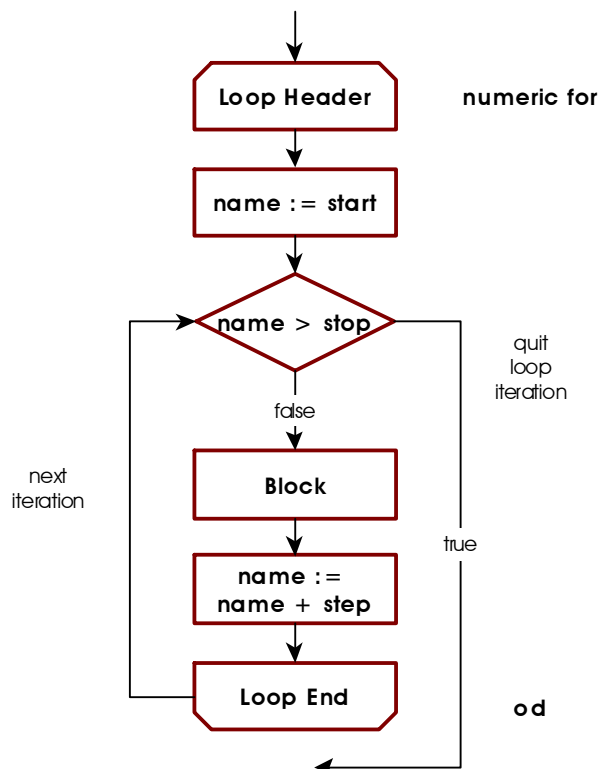
**for** loops are used if the number of iterations is known in advance. There are **for/to** loops for numeric progressions, and **for/in** loops for table and string iterations, see the next chapter.



## 5.2.2 for/to Loops

Let us first consider numeric **for/to** loops which use numeric values for control:

```
for name [from start] [to stop]
  [by step] do
    statements
od
```



*name*, *start*, *stop*, and *step* are all numeric values or must evaluate to numeric values.

The statement at first sets the variable *name* to the value of *start*. *name* is called the *control* or *loop variable*. If *start* is not given, the start value by default is +1.

When omitting the **to** clause, the loop iterates until the largest number representable on your platform has been reached. If left out, the *step* size is +1.

The **for** loop then checks whether  $start \leq stop$ . If so, it executes *statements* and returns to the top of the loop, increments *name* by *step* and then checks whether the new value is less or equal *stop*. If so, *statements* are executed again.

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

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

The control variable of a loop is always accessible to its surrounding block, so that you can use its value in subsequent statements. This rule applies only to **for/from/to**-loops with or without a **while**, **as** or **until** extension, but not to **for/in** loops

described below. Note that within procedures, the loop control variable is automatically declared local, while on the interactive level it is global.

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

The following rules apply to the value of the control variable after leaving the loop:

1. If the loop terminates normally, i.e. if it iterates until the stop value has been reached, then the value of the control variable will be its stop value *plus* the step size.
2. If the loop is left prematurely by executing a **break** statement<sup>13</sup> within the loop, or if a **for/while** loop is terminated because the **while** condition evaluated to **false** (see Chapter 5.2.8), then the control variable will be set to the loop's last iteration value before quitting the loop. There will be no increment with the loop's step size. The same applies to **for/as** and **for/until** loops (see Chapter 5.2.9).

Loops can count backwards if the step size is negative (see also the next chapter):

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

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

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

Agena automatically uses an advanced precision algorithm based on Neumaier summation if the step size is non-integral, e.g. 0.1, -0.01. This mostly prevents round-off errors, thus avoids that the loop stops before the last iteration value - the limit - has been reached and that iteration values with round-off errors are returned. You may switch Agena into Kahan-Ozawa or Kahan-Babuška summation mode to use extended round-off prevention by issuing the statement in a session:

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

or

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

---

<sup>13</sup> See Chapter 5.2.8 for more information in the **break** statement.

As a further measure to prevent a loop stopping before the stop limit has been reached, numeric **for** loops with fractional step sizes automatically increase the stop limit by the value of the constant **hEps**. If you pass a step size that is equal or less than **hEps**, Agenda now issues an error. You can entirely switch off this **math.Eps** to zero, but only by calling **environ.kernel**:

```
> environ.kernel(hEps = 0);
```

Kahan-Babuška summation may be more accurate than Kahan-Ozawa summation. The speed loss with both algorithms compared to Neumaier is around 20 percent or more.

If the step size is an integer, e.g. 1000, 1, -1.0, then Agenda will not use advanced precision to ensure maximum speed.

Tip: Use the **sumup** and **mulup** operators to approximate series and products, respectively, for they are around twice as fast as numeric **for** loops combined with **math.kbadd** for Kahan-Babuška summation (series) or internal 80-bit precision (products) to minimise round-off errors.

As syntactic sugar, Agenda accepts the **:=** assignment token in numerical **for** loops. In this case you must always give at least a start and stop value, the latter preceded by a **to** or **downto** token (see next chapter), and optionally accompanied by a **by** clause for the step size, and/or **while** or **until** conditions:

```
> for i := 1 to 10 while i < 5 do statements od;
> for i := 10 downto 1 by 0.5 do statements od
```

### 5.2.3 for/downto Loops

count from a **start** value *down* to a **stop** value, with a default countdown **step** size of (implicit minus) one. To count down, the optional **step** size should be positive.

```
for name from start downto stop [by step] do
    statements
od
```

### 5.2.4 for/in Loops over Tables

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

If **null** is passed after the **in** keyword, or if the value evaluates to **null**, then Agenda will not execute the loop and continue with the statement following it.

Let us first concentrate on table iteration.

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

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

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

There are two variations: When putting the token **keys** in front of the control variable, the loop iterates only over the keys of a table:

```
for keys key in tbl do  
    statements  
od
```

Example:

```
> for keys i in a do  
>     print(i)  
> od  
1  
2  
3
```

The other variation iterates on the values of a table only:

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

```
> for i in a do  
>     print(i)  
> od  
4  
5  
6
```

The control variables in **for/in** loops are always local to the body of the loop (as opposed to numeric **for** loops). You may assign their values to other variables if you need them later.

You should never change the value of the control variables in the body of a loop - the result would be undefined. Use the **copy** function to safely traverse any structure if you want to change, add, or delete its entries.

Because of the implementation of tables, please note that the keys in a table are not necessarily traversed in ascending order. You may want to iterate sequences or linked lists (see Chapter 6.28).

### 5.2.5 for/in Loops over Sequences and Registers

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

### 5.2.6 for/in Loops over Strings

If you want to iterate over a string character by character from its left to its right, you may use a **for/in** loop as well. All of the variations are supported.

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

for value in string do statements od

for keys value in string do statements od
```

The following code converts a word to a sequence of abstract vowel, ligature, and consonant place holders and also counts their respective occurrence:

```
> str := 'æfter';

> result := '';

> c, v, l -> 0;

> for i in str do
>   case i
>     of 'a', 'e', 'i', 'o', 'u' then
>       result &:= 'V';
>       inc v
>     of 'å', 'æ', 'ø', 'ö' then
>       result &:= 'L';
>       inc l
>     else
>       result &:= 'C'
>       inc c
>   esac
> od;

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

### 5.2.7 for/in Loops over Sets

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

```
> sister := {'swistar', 'sweastor', 'svasar', 'sister'}
> for i in sister do print(i) od;

svasar
swistar
sweastor
sister
```

You may try the other loop alternatives to see what happens.

### 5.2.8 for/in Loops over Procedures

The following procedure, called an iterator, returns a sequence of values multiplied by two. If `state = n`, then the procedure will return **null**, quitting the **for/in** iteration. Note that the iterator in its first result `n` returns the next value of the loop control variable `i`. We use `state` to hold the number of iterations we wish to perform. See Chapter 6 which describes procedures in detail.

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

In the following loop, 5 denotes the state and 0 the initial value.

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

Another means to iterate over procedures are closures, see Chapter 6.22. So far, here is just an example that you can use as a template for further experiments:

```
> iterate := proc(obj) is
>   local n := 0;      # with each call, counts up by one
>   return proc() is
>     inc n;
>     if n <= size obj then
>       return n, obj[n]
>     else
>       return null # quit iteration
>     fi
>   end
```

```
> end;

> f := iterate(seq(Pi, 2*Pi, 3*Pi));

> for i, j in f do
>   print(i, j)
> od;

1      3.1415926535898
2      6.2831853071796
3      9.4247779607694
```

You might also use the generic **ipairs** and **pairs** functions with **for/in** loops:

**ipairs** iterates table arrays, sequences, registers, strings and userdata that have an `'__index'` metamethod, in a standard way:

```
> for i, j in ipairs(s) do
>   print(i, j)
> od;

1      3.1415926535898
2      6.2831853071796
3      9.4247779607694

> d := numarray.double(3)

> for i to 3 do d[i] := i*Pi od

> for i, j in ipairs(d) do
>   print(i, j)
> od

1      3.1415926535898
2      6.2831853071796
3      9.4247779607694
```

To check whether a userdata features an `'__index'` entry in its associated metatable, just enter:

```
> getmetatable(d).__index:
procedure(01CE6DD0)
```

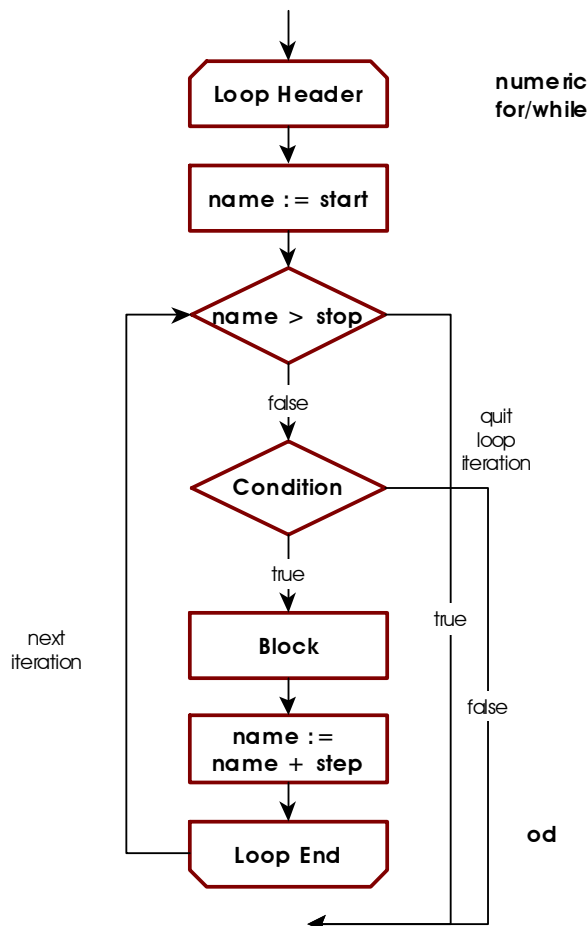
**pairs** allows to iterate all the keys and corresponding values of a dictionary, but as the following example shows, not surprisingly in a ``random`` fashion:

```
> t := [a = Pi, b = 2*Pi, c = 3*Pi]

> for i, j in pairs(t) do
>   print(i, j)
> od

a      3.1415926535898
c      9.4247779607694
b      6.2831853071796
```

Take in mind that **ipairs** and **pairs** are much slower than iterating structures directly.



### 5.2.9 for/while and for/until Loops

All flavours of **for** loops can be combined with a **while** condition. As long as the **while** condition is satisfied, the **for** loop iterates. To be more precise, before Agena starts the first iteration of a loop or continues with the next iteration, it checks the while condition to be **true** or any other value except **false**, **fail** or **null**. An example:

```

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

```

Regardless of the value of the **while** condition, the loop control variables are always initiated with the start values: in the summary frame below, with **for/to** loops, *a* is assigned to *i* (or 1 if the **from** clause is not given); *key* and/or *value* are assigned with the first

item in the table, set or sequence *struct* or the first character in string *string*. Likewise, the **until** condition quits a loop until it is satisfied.

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

The optional **while** and **until** clauses accept a simple assignment. In such a case, the right-hand side of the assignment is evaluated and stored to the left-hand side non-local name. The result of the evaluation is then checked and either the loop body is executed or not. Example:

```

> a := [10, 20, 4 ~ 30] # the table has no index 3

> for i to 4 while t := a[i] do # since a[3] evaluates to null,
>   # which is equal to false in this context, the loop quits with i = 3.
>   print(a[i], t)
> od
10      10
20      20

```



### 5.2.10 for/as & for/until Loops

As with the optional **while** clause, all flavours of **for** loops can be combined with an **as** or an **until** condition.

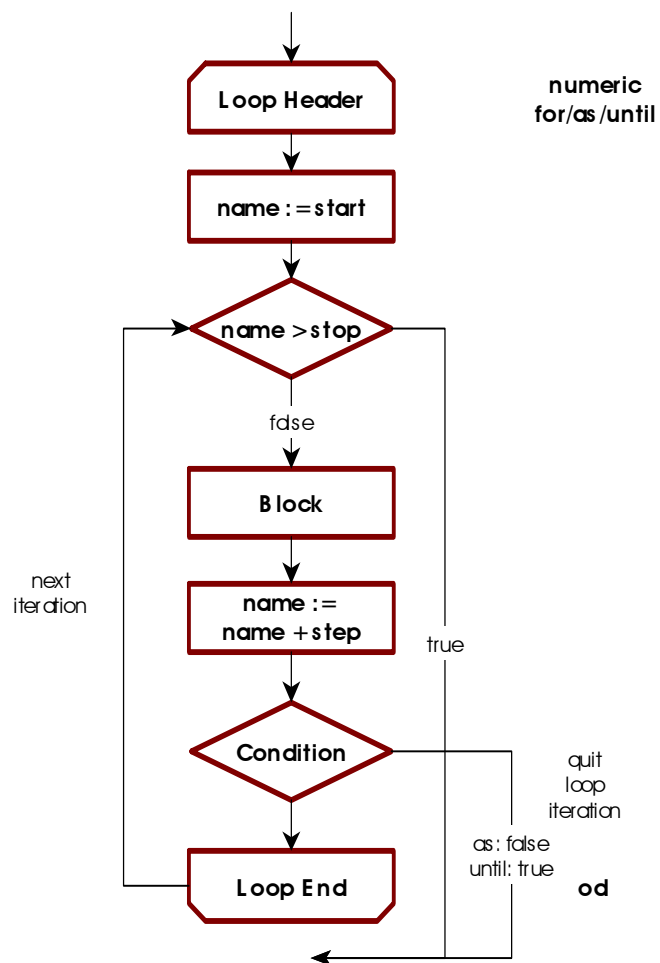
In these cases, a loop is always iterated at least once, and after the first iteration is completed, Agena checks the given condition and decides whether to start the next iteration or to leave the loop.

In the following example, the **for/as** loop starts with  $i=0$  and since the first check to the **as** condition results to **true**, the next iteration with  $i=1$  is conducted. The next check to the **as** condition results to **false**, thus the loop quits.

```
> for i from 0 do
>   print(i, 10^i)
> as 10^i < 10
0      1
1      10
```

The next loop iterates three times, until  $i=2$ , since only then the **until** condition becomes **true**.

```
> for i from 0 do
>   print(i, 10^i)
> until 10^i > 10
0      1
1      10
2      100
```

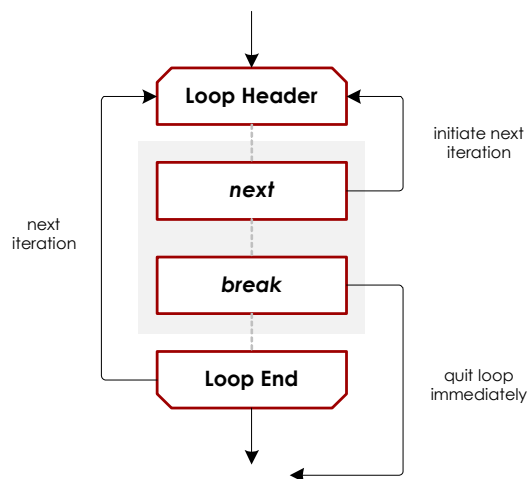


### 5.2.11 Loop Jump Control

Agena features statements to manipulate loop execution. **next** and **break** are applicable to all loop types, whereas **redo** and **relaunch** work in **for** loops only.

The **next** 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 next 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 next fi;
>   print(i)
> od;
1
2
4

```

```

> a := 0;

> while true do
>   inc a;
>   if a > 5 then break fi;
>   if a < 3 then next fi;
>   print(a)
> od;
3
4
5

```

There exists syntactical sugar for both the **next** and the **break** statements: instead of putting these statements into **if** clauses, just add the **when** or **unless** tokens along with a condition to the respective keyword.

```

> a := 0;
> while true do
>   inc a;
>   break when a > 5;
>   next when a < 3;
>   print(a)
> od;

```

```

> a := 0;
> while true do
>   inc a;
>   break unless a <= 5;
>   next unless a >= 3;
>   print(a)
> od;

```

Both flavours return:

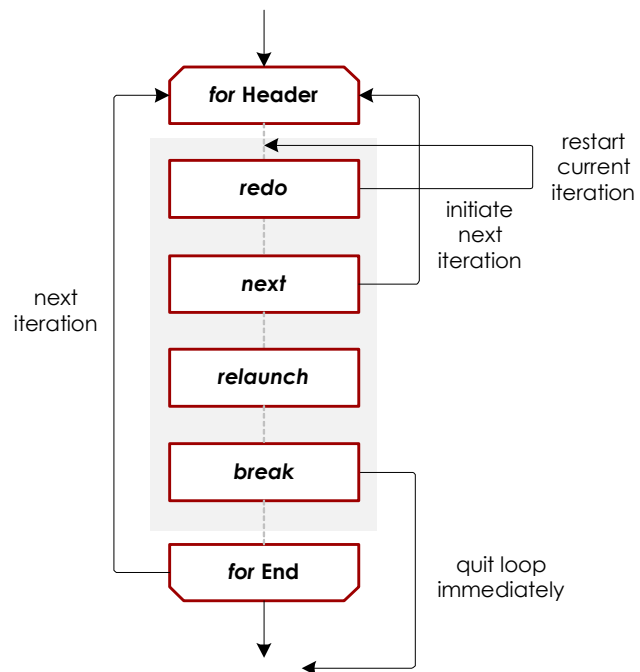
```
3
4
5
```

In **for/to** and **for/in** loops, the **redo** statement is similar to **next**: it jumps back to the beginning of the loop but does not change the loop control variable in **for/to** loops or the index/value control variables in **for/in** loops. Thus, it restarts the *current* iteration. At restart, it checks an optional **while** condition, if present.

```
> flag := true;

> for j in [10, 11, 12] do
>   print(j, flag);
>   if flag and j = 11 then
>     clear flag;
>     print(j, flag,
>       'jump back')
>     redo
>   fi
> until j > 12;

10    true
11    true
11    false    jump back
11    false
12    false
```



The **relaunch** statement completely restarts a **for/to** and **for/in** loop from its very beginning, i.e. resets the current control variable to its start value (**from** clause or first element, respectively).

```
> flag := true;

> for j in [10, 11, 12] do
>   print(j, flag);
>   if flag and j = 11 then
>     clear flag;
>     print(j, flag,
>       'restart')
>     relaunch
>   fi;
> until j > 12;

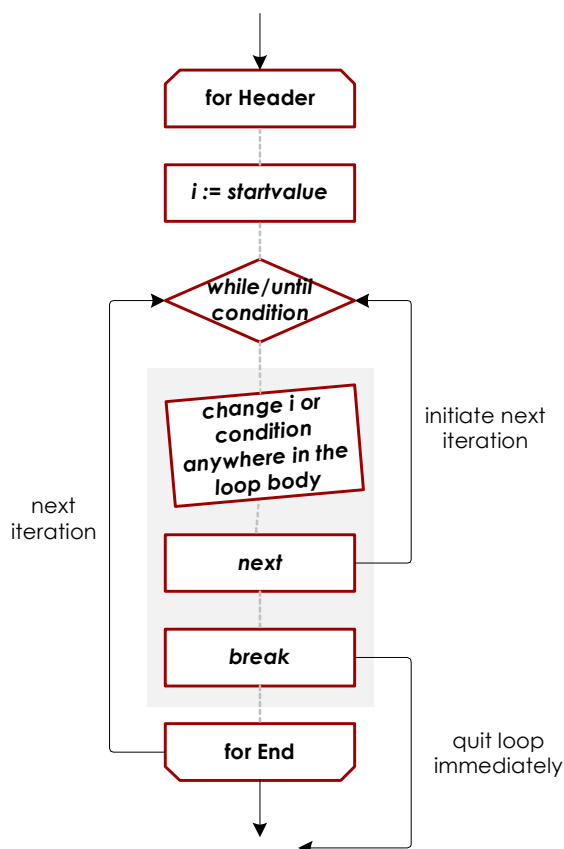
10    true
11    true
11    null    restart
10    null
11    null
12    null
```

### 5.2.12 Conditional for Loops

The conditional for loop initialises a new local control variable, checks a **while** or **until** condition and then executes the loop body. When the loop exits, the last value of the loop control variable is available in the block that is surrounding the loop.

**for**  $i := \text{value}$  (**while** | **until** | ,) *condition* **do** *statements* **od**

You may have to explicitly change the loop control variable in the loop block- or you might go into an infinite loop otherwise. Explicitly having to change it, however, gives you full control over - maybe varying - step sizes during a computation, for example when exploring highly-oscillatory functions with adaptive interval lengths.



As you can see in the diagram to the left, the `loop control variable` isn't a real one, as this loop variant actually is a while loop with an optional initialiser that does not need to be declared before and that also may not have any connection with the **while** or **until** condition at all.

Note that if you change the `loop control variable` in the loop body by a fractional value, the **while** or **until** condition might never be met due to accumulating round-off errors. See below for ways how to cope with that.

The loop body can include **redo**, **relaunch**, **next** and **break** statements. If you use **next**, **redo** or **relaunch**, you might change the loop control variable before or the loop might never finish.

Example with a **while** condition:

```

> for i := 1 while i <= 3 do print(i); i += 1 od
1
2
3

> i:
4

```

Instead of the **while** keyword, you might use a comma instead:

```
> for i := 1, i <= 3 do print(i); i++ od
1
2
3
```

Example with an **until** condition:

```
> for i := 1 until i = 4 do print(i); i++ od
1
2
3

> i:
4
```

The following two code snippets allow to compare standard for/from/while loops with conditional loops, both implementing the Mandelbrot set:

<pre>mandelbrot1 := proc(x, y, iter, radius) is   local c, z;   z := x!y;   c := z;   for i from 0 to iter while  z  &lt; radius do     z := square z + c   od;   return i end;</pre> <pre>mandelbrot1(0, 0.75, 256, 2): 33</pre>	<pre>mandelbrot2 := proc(x, y, iter, radius) is   local c, z;   z := x!y;   c := z;   for i := 0 while  z  &lt; radius do     z := square z + c; i++   od;   return i end;</pre> <pre>mandelbrot2(0, 0.75, 256, 2): 33</pre>
---	--

Concerning floating-point round-off errors, we first examine the result of the following statement:

```
> for i := -100, i <= 10 do
>   print(i)
>   i += 0.01
> od;
```

The loop stops with 9.9900000000141, around one step size short.

We could try a more elaborate approach using the `<` 'less-than' and `~=` approximate equality operators:

```
> for i := -100, i < 10 or i ~= 10 do
>   print(i)
>   i += 0.01
> od;
```

with the last iteration value of 10.000000000014. That is an error of 1.3999468251313e-011.

We can employ an accumulator that works with Kahan-Neumaier summation, automatically correcting round-off errors as best as possible. We first create an iterator and initialise it with -100:

```
> accu := math.accu(-100);
```

and use it as follows:

```
> for i := -100, i < 10 or i ~= 10 do
>   print(i)
>   i := accu(0.01); # add 0.01 to the accumulator
> od;
```

Now the loops stops at  $\sim 10.0$  with an error of  $1.7763568394003e-015$  only.

If you have a step size that remains always the same it is recommended to use numeric **for/from/to/by** loops instead of conditional **for** loops as the former conduct auto-adjustment and are much faster.

### 5.2.13 Scope I: scope and epocs

You can define the scope of local variables with the **scope/epocs** statement. Any variable declared local between the **scope** and **epocs** keywords exists only in this block, and they are not available outside of it:

<b>scope</b> <i>declarations and statements</i> <b>epocs</b>
--

An example:

```
> a := 2;

> scope
>   local b := 3; # b is local to the scope only
>   c := a*b      # c is available outside the block
> epocs;

> print(a, b, c);
2      null    6
```

Alternatively you can use the **begin** keyword to open a scope and **end** to close it.

### 5.2.14 Scope II: with Statement

The **with** statement allows to define a scope and assign one or more local variables in only one stroke. It is syntactic sugar to the scope statement only. The following example refers to the example in the preceding subchapter:

<b>with</b> <i>name<sub>1</sub>, ... := expr<sub>1</sub>, ...</i> <b>do</b> <i>declarations and statements</i> <b>od</b>
--

```
> a := 2;

> with b := 3 do # b is local, a and c are global
>   c := a*b
> od;

> print(a, b, c);
2      null      6
```

Assign multiple local variables, in this case two variables:

```
> a := 2;

> with b, c := 3, 4 do
>   d := a*b*c
> od;

> print(a, b, c, d);
2      null      6      24
```

### 5.2.15 with Statement for Dictionaries

The **with** statement can also unpack table values, indexed by string keys, declare them local and then access them in the respective block. After leaving the block, all the values listed between the **with** and **in** tokens are automatically written back to the table:

```
with key1 [, key2, ...] in tablename do
  statements
od
```

```
> zips := ['duedo' ~ 40210:40629,
>         bonn    = 53111:53229,
>         cologne = 50667:51149];

> with duedo, cologne in zips do # bonn has not been given here
>   print(duedo, bonn, cologne);
>   cologne := null; # cologne entry will be deleted from table zips
>   duedo := 40210:51149 # duedo entry in zips will be changed
>   # bonn entry will not be changed since not listed in the header
>   bonn := null
>   print(bonn, cologne, duedo)
> od;

40210:40629      null      50667:51149
null      null      40210:51149

> zips:
[bonn ~ 53111:53229, duedo ~ 40210:51149]
```

You may use simple brackets around a shortcut if it should evaluate to **null**, especially when it shall be used as a key in tables definitions.

Another flavour of the **with** statement has the following syntax:

```
with tablename do
    statements
od
```

Within the body of this variant, the table *tablename* can be referenced by just an underscore. It also allows to actively change values in *tablename*. Example:

```
> zips := [duedo = 4000, bonn = 5300]

> with zips do
>   print(_.bonn);
>   _.bonn := 53111
> od
5300

> zips:
[bonn ~ 53111, duedo ~ 4000]
```

### 5.2.16 Alternative to Closing Keywords

You can use the **end** token instead of the closing **fi**, **od**, **esac**, **yrt** and **epocs** keywords, or mix both.

Example:

```
> if os.system()[1] in {'SunOS', 'Windows', 'Linux', 'Darwin'} then
>   if environ.kernel().is32bit then
>     readlib('fractals');
>     readlib('gdi');
>     readlib('gzip');
>     a, b := gzip.deflate('agena programming language');
>     if [gzip.inflate(a, b)] <> ['agena programming language', 26] then
>       print('error in gzip.in\\deflate')
>     end;
>     try # provoke segfaults
>       for i from 0 to 100 do
>         gzip.inflate(a, i)
>       od
>     end
>   end;
>   to 100 do readlib('net') end # try crashing Agena at exit
> fi;
```



## Chapter **Six**

# Programming



## 6 Programming

Writing effective code in a minimum amount of time is one of the key features of Agena. Programmes are usually represented by procedures. The words `procedure` and `function` are used synonymously in this text.

### 6.1 Procedures

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

Writing procedures in Agena is quite simple:

```
procname := proc( [par1 [::type1] [, par2 [::type2], . . . ] ) [:: returntype] [is]
  [local name1 [, name2, . . .]];
  statements
end
```

All the values that a procedure shall process are given as *parameters* *par*<sub>1</sub>, etc. A function may have no, one or more parameters. A parameter may be succeeded by the name of a type (see Chapter 6.8.2), or a set of up to four types, that an argument must satisfy when the procedure is called.

If a type is given right after the parameter list, Agena checks whether the return of the procedure is of the given *returntype*, which may also be a user-defined type. The **is** keyword is optional.

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

Global variables are supported in Agena, as well. All values assigned on the interactive level are global, and you can also create global variables within a procedure. The values of global variables can be accessed on the interactive level and within any procedure.

A procedure may call other functions or itself. A procedure may even include definitions of further local or global procedures.

The result of a procedure will be returned through the **return** keyword which may be put anywhere in the procedure body, and which also immediately terminates execution of the procedure.

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

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

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

- if no **return** statement is given at all,
- if no values are given in the **return** statement.

The following procedure computes the factorial of an integer<sup>14</sup>:

```
> restart;

> fact := proc(n) is
>   # computes the factorial of an integer n
>   if n < 0 then return fail
>   elif n = 0 then return 1
>   else return fact(n-1)*n
>   fi
> end;
```

It is invoked using the syntax:

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

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

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

When calling a function recursively, instead of writing out its real name, you may use the **procname** keyword, which in runtime is substituted by the name with which the procedure was invoked:

```
> fact := proc(n) is
>   # computes the factorial of an integer n
>   if n < 0 then return fail
>   elif n = 0 then return 1
>   else return procname(n-1)*n
>   fi
> end;
```

A **when** clause can be added to a **return** statement that does not pass back any value including **null**. In this case, the execution of a function is being finished if the Boolean **when** condition has been satisfied, e.g. `return when x <> 0`. **return** can be combined with both a **when** and **with** clause - for example

```
> return when x <> 0 with true;
```

is syntactic sugar for

```
> if x <> 0 then
>   return true
> fi;
```

---

<sup>14</sup>The library function **fact** is much faster.

The **unless** keyword is the "opposite" to **when**, causing Agena *not* to return if the condition evaluates to **true**.

Last of all, procedures can alternatively be defined as follows:

```
[local] proc procname( [par1 [::type1] [, par2 [::type2], ... ] ) [:: returntype] [is]
  [local [constant] name1 [, [constant] name2, ...]];
  statements
end
```

Instead of the **proc** keyword, you can use the **procname** token. Thus, the factorial function can also be entered as follows:

```
> proc fact(n) is
>   if n < 0 then return fail
>   elif n = 0 then return 1
>   else return procname(n-1)*n
>   fi
> end;
```

## 6.2 Local Variables

The function above does not need local variables as it calls itself recursively. However, with large values for *n*, the large number of unevaluated recursive function calls will ultimately cause stack overflows. So we should use an iterative algorithm to compute the factorial and store intermediate results in a local variable.

A local variable is known only to the respective procedure and the block where it has been declared. It cannot be used in other procedures, the interactive Agena level, or outside the block where the local variable has been declared.

A local variable can be declared explicitly anywhere in the procedure body, but at least before its first usage. If you do not declare a variable as local and assign values later to this variable, then it will be global. Note that control variables in **for** loops are always implicitly declared local to either their surrounding (**for/to** loops) or inner block (**for/in** loops), so we do not need to explicitly declare them.

Local declarations come in different flavours:

```
local name1 [, name2, ...]
local [constant] name1 [, [constant] name2, ...] := value1 [, value2, ...]
local [constant] name1 [, [constant] name2, ...] -> value
local enum name1 [, name2, ...] [from value]
local key1 [, key2, ...] in tablename
```

In the first form, *name<sub>1</sub>*, etc. are declared local.

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

In the fourth form, *name<sub>1</sub>*, etc. are declared local and subsequently enumerated, i.e. assigned integers in ascending order, by default starting from 1, or the integer given in the optional **from** clause.

In the last form, table values are unpacked, equivalent to the assignment statement *key<sub>1</sub>*, *key<sub>2</sub>*, etc. := *tablename.key<sub>1</sub>*, *tablename.key<sub>2</sub>*, etc., with *key<sub>1</sub>*, *key<sub>2</sub>*, etc. being automatically declared local.

By passing the **constant** keyword in front of a variable name, a variable will become a constant that cannot be changed later in a session. This feature works in procedures only, not on the interactive level.

Let us write a procedure to compute the factorial using a **for** loop. To avoid unnecessary loop iterations when the intermediate result has become so large that it cannot be represented as a finite number, we also add a clause to quit loop iteration in such a case.

```
> fact := proc(n) is
>   if n < 0 then return fail fi;
>   local result := 1;
>   for i from 1 to n do
>     result := result * i
>     if not finite(result) then break fi
>   od;
>   return result
> end;

> fact(10):
3628800
```

Since *result* has been declared local it does not exist on the interactive level:

```
> result:
null
```

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

**create local** <structure> *name<sub>1</sub>* [, <structure> *name<sub>2</sub>*, ...]

where <structure> might be the keyword **table**, **set** or **sequence**. You can declare different local structures with one **create local** statement.

Two useful functions are **environ.globals** and **debug.unused** which determine global variable assignments inside procedures and declarations of unused local variables.

### 6.3 Global Variables

Global variables are visible to all procedures and the interactive level, such that their values can be queried and altered everywhere in your code.

Using global variables is not recommended. However, they are quite useful in order to have more control on the behaviour of procedures. For example, you may want to define a global variable `_EnvMoreInfo` that is checked in your procedures in order to print or not to print information to the user.

Global variables can be depicted with the **global** statement. It checks whether the given variable or variables have not been declared local before its execution and issues an error otherwise.

```
> fact := proc(n) is
>   if n < 0 then return fail fi;
>   local result := 1;
>   global _EnvMoreInfo;
>   for i from 1 to n do
>     result := result * i
>     if result = infinity then
>       if _EnvMoreInfo then print('Overflow !') fi;
>       break
>     fi
>   od;
>   return result
> end;
```

We should assign `_EnvMoreInfo` any value different from **null**, **fail** or **false** in order to get a warning message at runtime.

```
> _EnvMoreInfo := true;

> fact(10000):
Overflow !
infinity
```

### 6.4 Changing Parameter Values

You can change the values of procedure parameters within a procedure. Thus, an alternative to the **abs** operator might be:

```
> myAbs := proc(x) is
>   if x < 0 then
>     x := -x
>   fi;
>   return x
> end;

> myAbs(-1):
1
```

## 6.5 Optional Arguments

A function does not have to be called with exactly the number of parameters given at procedure definition. You may also pass less or more values. If no value is passed for a parameter, then it will be automatically set to **null** at function invocation. If you pass more arguments than there are actual parameters, excess arguments will be ignored.

For example, we can control whether a warning message is printed during function execution by passing an optional argument:

```
> fact := proc(n, warning) is
>   if n < 0 then return fail fi;
>   local result := 1;
>   for i from 1 to n do
>     result := result * i
>     if result = infinity then
>       if warning then print('Overflow !') fi;
>       break
>     fi
>   od;
>   return result
> end;

> fact(10000):
infinity
```

In this example, the option must be any value other than **null**, **false** or **fail** to get the effect.

```
> fact(10000, true):
Overflow !
infinity
```

A variable number of arguments can be passed by indicating them with a question mark in the parameter list and then querying them with the **varargs** system table in the procedure body. The **?** token can be used within in the procedure body as a shortcut to the **varargs** table.

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

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

You may determine the number of arguments *actually* passed in a procedure call by querying the system variable **nargs** inside the respective procedure. A variant of the above procedure might thus be:



```

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

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

```

Note: With OOP-style methods, **nargs** will also count the method itself.

Let us build an extended square root function that either computes in the real or complex domain. By default, i.e. if only one argument is given, the real domain is taken, otherwise you may explicitly set the domain using a pair as a second argument.

```

> xsqrt := proc(x, mode) is
>   if nargs = 1 or mode = 'domain':'real' then
>     return sqrt(x)
>   elif mode = 'domain':'complex' then
>     return sqrt(x + 0*I)
>   else
>     return fail
>   fi
> end;

> xsqrt(-2):
undefined

> xsqrt(-2, 'domain':'real'):
undefined

```

If the left-hand side value of the pair in a function call shall denote a string, you can spare the single quotes around the string by using the = token which converts the left-hand name to a string<sup>15</sup>.

```

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

```

You can mix optional arguments and the variable-arguments feature in parameter lists, with the question mark always the last item in the list:

```

> xsqrt := proc(x, mode, ?) is
>   ...
> end;

```

Finally, if you would like to define defaults for missing arguments, just use the binary **or** operator as shown below as it returns the first operand if it is non-**null**, and it returns the second operand if the first is **null**:

---

<sup>15</sup> If you need a Boolean equality check in a function call, such like  $f(a=b)$ , use the **isequal** function or the **==** operator, like  $f(\text{isequal}(a, b))$  Or  $f(a == b)$ .

```
> f := proc(x) is
>   x := x or 0;
>   return x
> end;

> f():
0
```

## 6.6 Passing Options in any Order

We can use variable arguments along with pairs in order to pass one or more optional arguments in any order.

```
> f := proc(?) is
>   local bailout, iterations := 2, 128; # default values
>   for i to nargs do
>     case left(?[i])
>       of 'bailout' then
>         bailout := right(?[i]);
>       of 'iterations' then
>         iterations := right(?[i]);
>       else
>         print 'unknown option'
>       esle
>     esac
>   od;
>   print('bailout = ' & bailout, 'iterations = ' & iterations)
> end;

> f();
bailout = 2      iterations = 128

> f('bailout':10);
bailout = 10     iterations = 128

> f('iterations':32, 'bailout':10);
bailout = 10     iterations = 32
```

Again, the quotes around the option name (the left-hand side of the pair) can be spared by giving the = token which converts the name to a string.

```
> f(bailout = 10, iterations = 32);
bailout = 10     iterations = 32
```

Sometimes, implementing checks on options may take a substantial amount of programming time, so please have a look at the **checkoptions**, **copyadd** and the **opt\*** functions which may save up to 20 % of code. You might consult Chapter 8 for further details.

## 6.7 Type Checking

Although Agena is untyped, in many situations you may want to check the type of a certain value passed to a function. Agena has four facilities for this:

1. the **type** operator determines the basic type of its argument;
2. the **typeof** operator returns a basic or user-defined type;

3. the `::` operator checks for a basic or user-defined type;
4. the `-:` operator checks whether a value is not of a given basic or user-defined type;

Basic or user-defined types can optionally be specified in the parameter list of a procedure by means of the preceding `::` token so that they will be checked at procedure invocation, see Chapter 6.8.2. Furthermore, the type or types of return of a procedure may be given right after the parameter list, see Chapter 6.8.3.

The following basic types are available in Agenda:

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

These names are reserved keywords, but with the exception of the **null** constant evaluate to strings so that they can be compared with the result of the **type** operator:

**type(value)**

```
> type(1):
number

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

If you want to check for the **null** type, put the **null** token in quotes:

```
> a := null;

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

The `::` and `-:` operators check whether their arguments are or are not of a specific type - or user-defined type - and return **true** or **false**. They are speed-optimised and around 20 % faster than comparing the return of the **type** operator with a type name.

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

Examples:

```
> 1 :: number:
true

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

In case of user-defined types, the type name must always be a string, in quotes. See Chapter 6.12 for more information. The `::` and `-:` operators can also isolate numbers further by passing the tokens `integer`, `posint`, `nonnegint`, `nonzeroint`, `positive`, `negative`, or `nonnegative`, see Chapter 6.8.2 for further information.

```
> -1 :: nonnegative:
false
```

The `::` and `-:` operators also accept a function for the right operand. If given, the operators call this function, which should evaluate to **true**, **false** or **fail**, with the left operand and finally return **true** or **false**. The operators interpret the call result **fail** as **false**. Examples:

```
> morethanten := <: x -> x :: number and x > 10 :>

> 11 :: morethanten:
true

> 11 -: morethanten:
false
```

Note that this does not work with the `::` type-check token in parameter lists of procedures, see Chapter 6.8.2.

## 6.8 Error Handling

### 6.8.1 The error Function

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

**error('error string')**

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

```
> fact('10'):
Error: number expected
```

```
Stack traceback:
  stdin, at line 3, at line 1
```

## 6.8.2 Type Checks in Procedure Parameter Lists

You may specify permitted types in the parameter list of a procedure by using double colons:

```
> fact := proc(n :: number) is
>   if n < 0 then return null
>   elif n = 0 then return 1
>   else return fact(n - 1)*n
>   fi
> end;
> fact('10'):
Error in stdin:
  invalid type for argument #1: expected number, got string.
```

This form of type checking is more than twice as fast as the **if/type/error** combination. If the argument is of the correct type, Agena executes the procedure, otherwise it will issue an error. Agena will also throw an error if the argument is not given:

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

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

```
> a := 1;

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

Furthermore, you may specify a set of one to five permissible *basic* types for any parameter with the set notation:

```
> sec := proc(x :: {number, complex}) is
>   return 1/cos(x)
> end;
```

Besides the basic types number, complex, string, table, set, pair, sequence and register, you can also pass the following keywords to further isolate numbers:

Keyword	Check for
integer	a number that represents a signed integer
posint	a number that represents a positive integer
nonnegint	a number that represents a non-negative integer
nonzeroint	a number that represents a non-zero integer
positive	checks for a positive number (float or integer)
negative	checks for a negative number (float or integer)
nonnegative	checks for a non-negative number (float or integer)

Note that in Agena there is only one type that represents floats and integers: type `number`. The above mentioned six numeric pseudo-types are only supported in parameter lists and by the `::` and `-:` operators.

Finally, there are three further pseudo-types:

- `anything` stands for any type, including `'null'`. If given in a parameter list, then Agena will check whether the corresponding argument of any type, even `'null'`, has been passed in a function call - if not, an error will be issued. The pseudo-type can also be passed as the right operand to the `::` and `-:` operators;
- `listing` identifies a table, sequence or register in the parameter list of a procedure. The type can be passed as the right operand to `::` and `-:`, as well.
- `basic` identifies a number, string, Boolean or **null**, and is recognised in parameter lists and the `::` and `-:` operators.

Examples that summarise these special types:

```
> proc f(x :: anything) :: listing is
>   return x
> end;

> f()
Error in stdin:
  missing argument #1 (of type anything).

Stack traceback:
  stdin, in `f`
  stdin, at line 1 in main chunk

> f(1)
Error in stdin at line 2:
  Error in `return`: result of type listing expected, got number.

Stack traceback:
  stdin, at line 2 in `f`
  stdin, at line 1 in main chunk

> f([1]):
[1]
```

### 6.8.3 Checking the Type of Return of Procedures

Agena can check whether all returns of a procedure are of one given type by specifying this return type right after its parameter list.

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

```
> fact(10):
3628800
```

If one of the returns is not of the return type, the procedure issues an error.

```
> fact := proc(n :: number) :: number is
>   if n < 0 then return undefined
>   elif n = 0 then return 1
>   else return 'don\'t know'
>   fi
> end;

> fact(10):
Error in stdin, at line 5:
  `return` value must be of type number, got string.

Stack traceback:
  stdin, at line 5, at line 1
```

The ‘virtual’ types `integer`, `posint`, `nonnegint`, `nonzeroint`, `positive`, `negative` and `nonnegative` can also be queried, see previous subchapter.

You can define up to five basic types that are allowed to be returned by putting them in curly brackets, just like in parameter lists:

```
> f := proc(x) :: {number, complex} is return 'a' end

> f()
In stdin at line 1:
  Error in `return`: unexpected type string in return.
```

If you would like to automatically check structures for proper content at function invocation, please have a look at the end of Chapter 6.19.

There are further functions for error handling:

### 6.8.4 The `assume` Function

**assume** checks a Boolean relation. If the relation is valid, it returns **true** and continues execution of the procedure. In case of an invalid relation, it bails out of the procedure and prints an error message. The second argument to **assume** is optional; if not given, the text ‘assumption failed’ is printed, and ‘error string’ otherwise.

**assume**(relation [, ‘error string’ ])

```
> assume(1 = 1, '1 is not 1'):
true      1 is not 1
```

```
> assume(1 <> 1, '1 is 1'):
Error in `assume`: 1 is 1.

Stack traceback: in `assume`
  stdin, at line 1 in main chunk
```

### 6.8.5 Trapping Errors with `protect/lasterror`

**protect** traps any error that might occur, but does not terminate a function call. In case of no errors, it returns all results of the call. But if there was an error, it returns the error message as a string and also sets the global variable **lasterror** to this error message. In case of a successful call, **lasterror** will always be **null**.

**protect** takes the name of the function to be executed as its first argument, and all its arguments *a*, *b*, etc. as optional arguments:

**protect**(*f* [, *a* [, *b*, ... ] ])

Thus, if a function has no arguments, simply pass the expression `protect(f)`.

```
> iszero := proc(x) is
>   if x <> 0 then
>     error('argument must be zero')
>   else
>     return true
>   fi
> end;
```

Now call `iszero` in protected mode:

```
> protect(iszero, 0):
true

> lasterror:
null

> protect(iszero, 1):
argument must be zero

> lasterror:
argument must be zero
```

To conveniently check whether an error occurred you might enter:

```
> protect(iszero, 0) = lasterror:
false

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



### 6.8.6 Trapping Errors with the try/catch Statement

Instead of intercepting errors with **protect** and **lasterror**, you may use the **try/catch** statement:

```
try
  statements1
[catch [in errvar then]
  statements2]
yrt
```

Any statements *statements<sub>1</sub>* - one or more - are put right after the **try** keyword. If an error occurs in one of these statements, Agena immediately will jump to the **catch** clause if present, ignoring any subsequent statements in *statements<sub>1</sub>*. If there is no **catch** clause, execution will immediately continue with the statement right after the **yrt** token, regardless of whether an error occurred or not, also ignoring all subsequent commands in *statements<sub>1</sub>*.

If a **catch** clause is given, then in case of an error the error message will be stored to the local variable *errvar*, and after that all the statements *statements<sub>2</sub>* following the **then** keyword are processed. *errvar* does not need to be declared, it is implicitly local to the **catch** clause only. You may also do without specification of an error variable - in this case the error message is automatically stored to the local **lasterror** variable, and the **in** and **then** keywords must be left out.

Examples:

```
> try
>   error('Oops !');
>   print('Invalid index !')
> yrt;
```

As shown above, due to the immediate jump out of the **try** body, the **print** function is not called. In the next example, the error message is stored to the variable *message*, and in the **catch** clause it is then printed at the console.

```
> try
>   error('Oops !');
>   print('Invalid index !')
> catch in message then
>   print('The error was: ' & message);
> yrt;
The error was: Oops !

> message:
null
```

Now we do not specify an error variable in the **catch** clause:

```
> try
>   error('Oops !');
>   print('Invalid index !')
> catch
>   print('The error was: ' & lasterror);
> yrt;
The error was: Oops !
```

### 6.8.7 Trapping Errors with pre and post Clauses

Instead of writing long special error treatment code when checking arguments or the return of a function, you may use the **pre** and **post** clauses:

The **pre** clause, placed right before the **is** keyword, checks a condition and issues an error if it is not met:

```
> golden := proc(n :: number)      # approximation of golden ratio
>   pre isint(n) and n > -1 is    # if n < 0 or float, quit with an error
>   if n = 0 then return 1 fi;
>   return 1.0 + 1.0/procname(n - 1);
> end;

> golden(-1):
In stdin at line 2:
  Error in pre-condition: posture not satisfied.
```

It is faster than checking arguments with calls to the **assume** function.

The **post** clause in **return** statements checks a condition and issues an error if it is not met:

```
> proc(x :: number) is
>   [...]
>   # issue an error if x <> 1, and return x otherwise
>   return post x <> 1 with x
> end;
```

A function can include both **pre** and **post** conditions.

## 6.9 Multiple Returns

As stated before, a procedure can return no, one, or more values. Just specify the values to be returned:

```
> f := proc() is
>   a := 2;
>   return 1, a
> end;

> f():
1      2
```

There are two ways to refer to these multiple returns in subsequent statements. If you assign the return to only one variable, e.g.

```
> m := f():
1
```

the second return is lost, so enter:

```
> m, n := f();
> m:
1
> n:
2
```

A function may return a variable number of values, so it might be useful to put them in a sequence, register or table:

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

Sometimes a procedure shall return the first result of a computation only. In this case, put the call that results into multiple returns into brackets. **math.fraction** returns three values: the numerator, the denominator, and the accuracy, in this order. Let us write a numerator function that only returns the first result of **math.fraction**.

```
> numerator := proc(x :: number) is
>   return (math.fraction(x))
> end;
> numerator(0.1):
1
```

The **ops** function returns all its arguments after argument number index, an integer.

**ops(index, arg<sub>1</sub> [, arg<sub>2</sub>, ... ] )**

The following statement determines the denominator and the accuracy.

```
> ops(2, math.fraction(0.1)):
10      0
```

To return only the first result, the denominator, put the call to **ops** in brackets.

```
> denominator := proc(x :: number) is
>   return (ops(2, math.fraction(x)))
> end;
> denominator(0.1):
10
```

**unpack** returns all elements in a table or sequence:

```

> squared := proc(t :: table) is
>   local result := <: x -> x^2 :> @ t;
>   return unpack(result)
> end;

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

```

Alternatively, **unpack** accepts the positions of the first to the last element to be returned as its second and third argument. If only the second argument is given, all elements in a structure from the given position up to the end are passed back.

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

```

> squared := proc(t :: table, ?) is
>   local result := <: x -> x^2 :> @ t;
>   return unpack(result, unpack(?))
> end;

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

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

```

## 6.10 Procedures that Return Procedures

Besides returning numbers, strings, tables, etc., procedures can also return procedures. As an example, the function `polygen`

```

> polygen := proc(?) is
>   local s := seq(unpack(?));
>   return proc(x) is
>     local r := bottom(s);
>     for i from 2 to size s do
>       r := r*x + s[i]
>     od;
>     return r
>   end
> end;

```

returns a procedure that evaluates a polynomial of degree  $n$  from the given coefficients  $c_n, c_{n-1}, \dots, c_2, c_1$ :

$$\text{<: (x) -> } c_n * x^{n-1} + c_{n-1} * x^{n-2} + \dots + c_2 * x + c_1 \text{ :>}$$

In the following example, `polygen` creates the polynomial  $3x^2 - 4x + 1$  as a procedure.

```

> f := polygen(3, -4, 1)

> f(2):
5

```

## 6.11 Shortcut Procedure Definition

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

```
<: [(] [par1 [:: type1] [, par2 [:: type2], ...]] [)] -> expr1 [, expr2, ...] :>

      <: [(] [par1 [:: type1] [, par2 [:: type2], ...]] [)]
      [ with var1 [, ...] := val1 [, ...] ] -> expr1 [, expr2, ...] :>
```

As you see, optional basic and user-defined types can be specified in the parameter section.

Let us define a simple factorial function.

```
> fact := <: (x :: number) -> exp(lngamma(x + 1)) :>;

> fact(4):
24
```

Brackets around parameters are optional if at least one parameter is given, even if you specify types.

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

> isInteger(1):
true

> isInteger(1.5):
false

> one := <: () -> 1 :>; # with no parameters, use empty bracket pair
```

Optional arguments and the ? notation are supported.

One or more local variables can be defined by the **with** clause put in front of the expression that computes the result:

```
> fact := <: (x :: number)
>   with n := 1
>   -> exp(lngamma(x + n)) :>;

> fact(4):
24
```

You can chain multiple assignment statements (two ore more) by separating them with semicolons:

```
> f := <: x, y with t := 1; u, v := t + 1, t + 2 -> print(x, y, t, u, v) :>
1       2       1       2       3
```

Short-cut procedures can return multiple results:

```
> f := <: x -> x, x+1, x+2 :>
```

```
> f(0):
0      1      2
```

Alternatively you can use **def** or **define**, both in expressions and as a statement. In this case you can define a function with zero, one or more parameters, but you can return one result only:

```
> sum := define x, y -> x + y;
```

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

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

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

The **with** clause defining one or more local variables is supported, multiple assignment, too, and one or more parameters can be optionally put in brackets, too:

```
> sum := define (x, y) with n := 1 -> x + y + n;
```

```
> sum(1, 2):
4
```

```
> define sum(x, y) with m, n := 10, 20 -> x + y + m + n
```

```
> sum(1, 2):
33
```

Required types may be given, as well:

```
> sum := define x :: number, y :: number with n := 1 -> x + y + n;
```

With zero parameters, use empty brackets:

```
> one := define () -> 1;
```

```
> one():
1
```

Contrary to the **<: ... :>** notation, **def** and **define** allow for one return expression only, so something like

```
> f := <: x, y -> x + y, 10 :>
3      10
```

will not work as expected:

```
> f := def x, y -> x + y, 10
```

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

## 6.12 User-Defined Procedure Types

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

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

User-defined procedures can be queried with the **typeof** operator which returns a string.

```
> f := <: x -> 1 :>;
> settype(f, 'constant');
> typeof(f):
constant
> type(f): # only returns the basic type
procedure
```

The **::** and **-:** operators can also validate a user-defined procedure type. Pass the name of the user-defined type as a string:

$\text{proc}_1 :: \text{'your\_proctype'}$   
 $\text{proc}_1 -: \text{'your\_proctype'}$

```
> f :: 'constant':
true
> f -: 'constant':
false
```

Note that the **type** operator only checks for basic types.

An alternative to **typeof** is the **gettype** function. If a user-defined type has been set for a value, then it will return its name as a string, otherwise, it will return **null**.

If you want to check whether user-defined types have been passed to a procedure, use the double colon notation in the parameter list.

Suppose you have defined a type called `triple`:

```
> t := [1, 2, 3]
> settype(t, 'triple')
```

```

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

> sum(t):
6

```

### 6.13 Scoping Rules

In Agena, variables live in blocks or ‘scopes’. A block may contain one or more other blocks. A local variable is visible only to the block in which it has been declared and to all blocks that are part of this block. Thus, variables declared local in inner blocks are not accessible to the outer blocks or outside the procedure in which they are hosted.

Procedures, **if**- and **case**-statements, **while**-, **do**- and **for**-loops create blocks, or more precisely, a block resides between:

1. **then** and **elif**, **else** or **fi** keywords - in **if** statements;
2. **then** and **of**, **else** or **esac** keywords - in **case** statements;
3. **do** and **as** - in **do/as** loops;
4. **do** and **od** - in **for** and **while** and **do/od** loops;
5. **is** and **end** - in procedures;
6. **scope** and **epocs**, **begin** and **end** - in **scope/begin** blocks (including the **with** statement; see below).

As an example, variables declared as local in the **then** clauses of an **if**-statement live only in the respective **then** part. The same applies to variables declared locally in **else** clauses.

```

> f := proc(x) is
>   if x > 0 then
>     local i := 1; print('inner', i)
>   else
>     local i := 0; print('inner', i)
>   fi;
>   print('outer', i) # i is not visible
> end;
> f(1);
inner  1
outer  null

```

Variables declared as local in **for**- or **while**-loops are only accessible in the bodies of these loops. The loop control variables of **for/to**-loops are automatically declared local to their surrounding block, while control variables of **for/in**-loops are implicitly declared local to the respective loop bodies.

```

> f := proc(x) is
>   while x < 2 do
>     local i := x
>     inc x
>     print('inner', i)
>   od;
>   print('outer', i) # i is not visible

```



```
> end;

> f(1);
inner    1
outer    null
```

A special scope can be declared with the **scope** and **epocs** statements:

```
scope
    declarations & statements
epocs
```

Alternatively, a scope may start with the **begin** keyword, finalised by **end**.

The next example demonstrates how it works:

```
> f := proc() is
>   local a := 1;
>   scope
>     local a := 2;
>     writeline('inner a: ', a);
>   epocs;
>   writeline('outer a: ', a);
> end;

> f()
inner a: 2
outer a: 1
```

The **scope** statement can also be used on the interactive level to execute a sequence of statements as one unit. Compare

```
> print(1);
1

> print(2);
2

> print(3);
3
```

with

```
> scope
>   print(1);
>   print(2);
>   print(3)
> epocs;
1
2
3
```

### 6.14 Access to Loop Control Variables within Procedures

As already mentioned, the control variable of a **for/to** loop is always local to the body surrounding the loop.

```
> mandelbrot := proc(x, y, iter, radius) is
>   local i, c, z;
>   z := x!y;
>   c := z;
>   for i from 0 to iter while abs(z) < radius do
>     z := z squareadd c # = z^2 + c
>   od;
>   return i # return the last iteration value
> end;
```

The procedure counts and returns the number of iterations a complex value  $z$  takes to escape a given radius by applying it to the formula  $z = z^2 + c$ .

```
> mandelbrot(0, 0, 128, 2):
129
```

The following example demonstrates that local variables are bound to the block in which they have been declared.

```
> f := proc() is
>   local i;
>   for i to 3 do
>     local j;
>     for j to 3 do od;
>     print(i, j)
>   od;
>   print(i, j)
> end;

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

### 6.15 Sandboxes

By default, every procedure has access to the full Agenda environment, i.e. to all of Agenda's functions, packages, and all the other values. You might want to limit this access, for example if one of your procedures offers services on the Internet, or you want a procedure maintain its own environment.

Here, the **environ.setfenv** function comes into play. It initialises the environment a function can use.

Example 1: Give access to all functions except the **os** package.

First copy Agenda's environment represented by the system table **\_G** to a new table so that altering this new table will not effect Agenda's normal environment:

```
> _newG := copy(_G); # copy can also duplicate cycles like _G
```

Delete the **os** package from this new environment:

```
> delete os from _newG;
```

Define a function that tries to determine the current working directory:

```
> curdir := proc() is
>   return os.chdir()
> end;
```

Set the environment excluding the **os** package:

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

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

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

Example 2: Give access only the specific functions.

Let us redefine curdir: it will only access a redefined **print** function and all of the functions of the **os** package. curdir cannot call any other function.

```
> curdir := proc() is
>   print(os.chdir())
> end;

> environ.setfenv(curdir,
>   ['print' ~ <: x -> print('cwd is ' & x) :>, 'os' ~ os])

> curdir():
cwd is C:/agenda/src
```

To determine the current environment used by a function, use **environ.getfenv**:

```
> environ.getfenv(curdir):
[os ~ (...), print ~ procedure(01D4BA18)]
```

Please see Chapter 14.2 (**environ.getfenv**, **environ.setfenv**, **environ.isselfref**) for further features.

To hide data in a sandbox, please have a look at registers - explained in Chapter 4.15.

## 6.16 Altering the Environment at Run-Time

Besides using a special environment (see preceding subchapter), a procedure can also create new variables and put them into Agenda's standard environment.

Why should one do so ? Consider the **utils.decodexml** function. It converts an XML string into a table consisting of key-value pairs, the keys being the XML tags, and the values the corresponding data. XML allows to use name spaces, so that tags might look like `<soap:body>`, etc.

So, XML data like

```
> str := '<soap:body>
>   <orderid>123</orderid>
> </soap:body>'
```

is converted to

```
> order := utils.decodexml(str):
[soap_body ~ [orderid ~ 123]]
```

To read the order number, one might just enter:

```
> order.soap_body.orderid:
123
```

Unfortunately, especially the SOAP standard allows one to define ones own name space, so that the following is also equivalent and valid XML data:

```
> str := '<s:body>
>   <orderid>123</orderid>
> </s:body>'

> order := utils.decodexml(str):
[s_body ~ [orderid ~ 123]]
```

In this case you would have to write a new statement to get the order ID since fetching it with

```
> order.soap_body.orderid:
Error in stdin, at line 1:
  attempt to index field `soap_body` (a null value)
```

will not work. Fortunately, Agena stores all values in the **\_G** system table, with its keys being strings representing the variable names, and the entries the values of the these variables. So flexible code to read data from XML code featuring different name spaces might look like this:

```
> str := '<s:body>
>   <orderid>123</orderid>
> </s:body>'

> order := utils.decodexml(str):
[s_body ~ [orderid ~ 123]]

> tag := tables.indices(order)[1]:
s_body
```

```
> prefix := tag[1 to ('_' in tag) - 1]:
s

> _G['order'][prefix & '_body'].orderid:
123
```

Likewise, defining new variables within code can be done like this:

```
> _G['jpl'] := ['Jet Propulsion Laboratory']

> jpl:
[Jet Propulsion Laboratory]
```

## 6.17 Packages

### 6.17.1 Writing a New Package

Let us write a small utilities package called `helpers` including only one main and one auxiliary function. The main function shall return the number of digits of an integer.

Package procedures are usually stored to a table, so we first create a table called `helpers`. After that, we assign the procedure `ndigits` and the auxiliary `aux.isInteger` function to this table.

```
> create table helpers, helpers.aux;

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

> helpers.ndigits := proc(n :: number) is
>   if not helpers.aux.isInteger(n) then
>     error('Error, argument is not an integer')
>   fi;
>   return if n = 0 then 1 else entier(ln(abs(n))/ln(10) + 1) fi
> end;
```

Now we can use our new package.

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

> helpers.ndigits(-10):
2

> helpers.ndigits(.1):
Error, argument is not an integer

Stack traceback: in `error`
  stdin, at line 3, at line 1
```

To save us a lot of typing, we can assign a short name to this table procedure.

```
> ndigits := helpers.ndigits;
```

```
> ndigits(999):
3
```

Save the code listed above to a file called `helpers.agn` in a subfolder called `helpers` in the Agena main directory. In order to use the package again after you have restarted Agena, use the **run** function and specify the full path.

```
> restart;

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

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

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

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

### 6.17.2 The initialise Function

The **initialise** function, besides loading the package in a convenient way, automatically assigns short names to all package procedures so that you may use the shortcuts instead of the fully written function names.

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

```
libname &:= ';d:/agena/helpers';
```

Alternatively, you may save the `helpers.agn` file into the `lib` folder of your Agena distribution if you do not want to modify **libname**.

Now in the interactive level, type:

```
> restart;
```

**libname** and some few other system variables are not reset by the **restart** statement because **restart** deliberately does not touch the contents of these specific system variables.

```
> initialise 'helpers'
ndigits

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

You may also want **with** to print a start-up notice at every package invocation by assigning a string to the table field ``packagename.initstring``. Put the following line

into the `helpers.agn` file after the **create table** statement, save the file and restart Agena:

```
> helpers.initstring := 'helpers v1.0 as of June 11, 2013\n\n';
> restart;
> initialise 'helpers'
helpers v1.0 as of June 11, 2013

ndigits
```

Since you may not want that short names are set for certain, especially auxiliary functions, their procedure names should be defined as follows: ``packagename.aux.procedurename``, e.g. `helpers.aux.isInteger`.

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

```
create table helpers, table helpers.aux;

helpers.initstring := 'helpers v1.0 as of June 11, 2013\n\n';

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

helpers.ndigits := proc(n :: number) is
    if not helpers.aux.isInteger(n) then
        error('argument is not an integer')
    fi;
    if n = 0 then
        return 1
    else
        return entier(ln(abs(n))/ln(10) + 1);
    fi;
end;
```

Save the file again and restart Agena.

```
> restart;
> initialise 'helpers'
helpers v1.0 as of June 11, 2013

ndigits
```

You can also define a package initialisation routine. It will automatically be run by the **initialise** statement after the package has been found and initialised successfully. The name of the initialisation routine must be of the form ``packagename.aux.init``, e.g.:

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

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

Instead of using **initialise** to load a package, you may use the **import/alias** statement - see Chapter 3.18 - so

```
> initialise 'helpers';
```

is equivalent to

```
> import helpers alias;
```

## 6.18 Remember Tables

Agena features remember tables which store the results of previous calls to Agena or C library procedures or contain a list of predefined results, or both. If a function is called again with the same argument(s), then the corresponding result will be returned from the table, and the procedure body is not executed, resulting in significantly better execution times. Remember tables are called *rtables* or *rotables* for short. They can hold up to 20,000 entries, discarding previous results when exceeding this limit.

All functions to create, modify, query, and delete remember tables are available in the **rtable** package.

There are two types of remember tables:

- Standard Remember Tables, called ``rtables``, that can be automatically updated by a call to the respective function; they may be initialised with a list of precomputed results (but do not need to).
- Read-only Remember Tables, called ``rotables``, that cannot be updated by a call to the respective function. Rotables should be initialised with a list of precomputed results.

### 6.18.1 Standard Remember Tables

A standard remember table is suited especially for recursively defined functions. It may slow down functions, however, if they have remember tables but do not rely much on previously computed results.

By default, no procedure contains a remember table. It must explicitly be created either by including the **feature reminisce** statement as the very first line in a procedure body, or by calling the **rtable.init** function right after the procedure has been defined. A remember table may optionally be filled with default values with the **rtable.put** function. Since those functions are very basic, a more convenient facility is the **rtable.remember** function which will exclusively be used in this chapter.

In order for an rtable to be automatically updated, the respective function must return its result with the **return** statement (which may sound profane). If a function is



called with arguments that are not already known to the remember table, then the **return** statement adds these arguments and the corresponding result or results to the rtable.

Let us first try the **feature reminisce** variant, which may suffice in most cases. Just add this statement right after the **is** token in a procedure that computes Fibonacci numbers:

```
> fib := proc(n) is
>   feature reminisce; # creates a read-write remember table
>   if n = 0 or n = 1 then return n fi; # exit conditions
>   return procname(n - 2) + procname(n - 1)
> end;

> fib(50):
20365011074
```

Now let us use the functions of the **rtable** package to administer remember tables.

Two examples: We want to define a function  $f(x) = x$  with  $f(0) = \text{undefined}$ .

First a new function is defined without using the **feature reminisce** phrase:

```
> f := proc(x) is return x end;
```

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

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

The rtable has now been created and a default entry add to it so that calling f with argument 0 returns **undefined** and not 0.

```
> f(1):
1

> f(0):
undefined
```

If the function is redefined, its remember table is destroyed, so you may have to initialise it again.

Fibonacci numbers, as already shown above, can be implemented recursively and run with astonishing speed using rtables.

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

The call to **assume** assures that *n* is always non-negative and serves as an ‘emergency brake’ in case the remember table has not been set up properly.

The *rtable* is being created with two default values:

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

If we now call the function,

```
> fib(50):
20365011074
```

the contents of the *rtable* will be:

```
> remember(fib):
[[22] ~ [28657], [39] ~ [102334155], [17] ~ [2584], [5] ~ [8], [27] ~
[317811], [50] ~ [20365011074], [3] ~ [3], [0] ~ [1], [46] ~ [2971215073],
[41] ~ [267914296], [1] ~ [1], etc.]
```

If a function has more than one parameter or has more than one return, **remember** requires a different syntax: The arguments and the returns are still passed as key~value pairs. However, the arguments are passed in one table, and the returns are passed in another table.

```
> f := proc(x, y) is
>   return x, y
> end;

> remember(f, [[1, 2] ~ [0, 0]]);

> a, b := f(1, 2);

> a:
0

> b:
0
```

Please check Chapter 14.4 for more details on their use.

### 6.18.2 Read-Only Remember Tables

If you do not want a function updating its remember table each time it is called with new arguments and results, you may use a read-only remember table, called ‘*rotable*’ for short. Rotables are initialised with a list of precomputed results.

The function itself cannot implicitly enter new entries to its remember table via the **return** statement; it can only do so via a call to the **rtable.put** function or a utility that is based on **rtable.put**, called **rtable.defaults**. This gives you full control on the contents and the amount of data stored in a remember table - and thus on the speed of your procedure.

Assume you want to define a procedure that computes factorials  $n!$ , and that does not compute the results for  $n < 11$ , but retrieves the results from a rotatable instead.

A function might look like this:

```
> fact := proc(x :: number) is
>   if x :: nonnegint then # is x an integer and non-negative ?
>     return exp(lngamma(x + 1))
>   else
>     return undefined
>   fi
> end;
```

The **defaults** function can set up the rotatable and enter precomputed values into it.

```
> # set precompiled results for 0! to 10! to fact
> defaults(fact, [
>   0~1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800
>   ]);
```

The factorial function is significantly faster when called with arguments that are in the rotatable than if there would be no such value cache, because it would have to re-compute the results instead of just reading them.

Let us look into the remember table:

```
> defaults(fact):
[[2] ~ [2], [1] ~ [1], [8] ~ [40320], [9] ~ [362880], [10] ~ [3628800],
[0] ~ [1], [4] ~ [24], [5] ~ [120], [6] ~ [720], [3] ~ [6], [7] ~ [5040]]
```

You can also easily add further argument ~ result pairs with the **table.defaults** function:

```
> defaults(fact, [11 ~ 39916800]);
> defaults(fact):
[[2] ~ [2], [1] ~ [1], [8] ~ [40320], [9] ~ [362880], [10] ~ [3628800], [0]
~ [1], [11] ~ [39916800], [4] ~ [24], [7] ~ [5040], [6] ~ [720], [3] ~ [6],
[5] ~ [120]]
```

A read-only remember table can be deleted by passing **null** as a second argument to **defaults**.

### 6.18.3 Functions for Remember Table Administration

For completeness, here is a list of all the functions to administer remember tables:

Procedure	Details
<b>rtable.forget</b> (f)	Empties the remember table of function $f$ but does not delete the table so that it will continue collecting results with the next call to $f$ . Read-only remember tables cannot be emptied. The memory previously occupied by cached function arguments and results can be reused for other purposes
<b>rtable.get</b> (f)	Returns the remember table of function $f$ .
<b>rtable.init</b> (f)	Initialises a standard remember table for the function $f$ .
<b>rtable.roinit</b> (f)	Initialises a read-only remember table for the function $f$ .
<b>rtable.put</b> ( f, [arguments], [returns])	Adds function argument(s) and the corresponding return(s) to the remember table of procedure $f$ .
<b>rtable.purge</b> (f)	Deletes the remember table of function $f$ entirely. The function empties the remember table before deleting it. It also enforces an immediate garbage collection. If you want to use a new remember table with the function, you have to initialise it with <b>rtable.init</b> or <b>rtable.roinit</b> again.
<b>rtable.mode</b> (f)	Returns the string 'rtable' if a function $f$ has a standard remember table, 'rotable' if it has a read-only remember table, and 'none' if it has no remember table at all.

Table 18: Functions for administering remember tables

## 6.19 Overloading Operators with Metamethods

One of the many useful features inherited from Lua 5.1 are metamethods which provide a means to use existing operators with tables, sets, sequences, registers, pairs and userdata.

For example, complex arithmetic can be implemented with sequences and metamethods so that you can apply various operators on the former, such as `+` or **abs**. Adding functionality to existing operators is also known as 'overloading'.

Let's start with an easy example. We will define a constructor to produce complex values and three metamethods for adding complex values with the `+` operator, calculating their absolute value with the standard **abs** operator and pretty-printing them at the console.

### 6.19.1 First Example: Constructor, Read & Write Operations, Prettyprinter

Let's store a complex value  $z = x + yi$  to a sequence of size 2. The real part is saved to position 1 of the sequence and the imaginary part to position two.

```
> cmplx := proc(re :: number, im :: number) is
>   create local sequence z(2);
>   insert re, im into z;
>   return z
> 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 pretty, so we would like Agenda to print `cmplx(0, 1)` instead of `seq(0, 1)`. This can be easily done with the **settype** function:

```
> cmplx := proc(re :: number, im :: number) is
>   create local sequence z(2);
>   insert re, im into z;
>   settype(z, 'cmplx');
>   return z
> end;

> cmplx(0, 1):
cmplx(0, 1)
```

Adding two complex values does not work yet, for we have not yet defined a proper metamethod.

```
> cmplx(0, 1) + cmplx(1, 0):
Error in stdin, at line 1:
  attempt to perform arithmetic on a sequence value
```

Metamethods are put into dictionaries, called ``metatables``. Their keys, which are always strings, denote the operators to be overloaded, the corresponding values are the procedures that are to be called when the operators are applied to tables, sets, sequences (which are used in this example), registers, pairs and userdata. See Appendix A2 for a list of all available method names. To overload the `+` operator we use the `'__add'` key.

We put the metamethod into table `cmplx_mt`, but any name will do:

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

Next, we will store this metatable `cmplx_mt` to the sequence that will hold the complex value, by calling the **setmetatable** function.

```
> cmplx := proc(re :: number, im :: number) is
>   create local sequence z(2);
>   insert re, im into z;
>   settype(z, 'cmplx');
>   setmetatable(z, cmplx_mt);
>   return z
> end;
```

Try it:

```
> cmplx(0, 1) + cmplx(0, 1):
cmplx(0, 2)
```

Add a new method to compute the absolute value of complex numbers (modulus) by overloading the **abs** operator.

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

The metatable now contains two methods.

```
> cmplx_mt:
[__add ~ procedure(004A64D0), __abs ~ procedure(004D2D30)]

> z := cmplx(1, 1);

> abs(z):
1.4142135623731
```

It would be fine to print complex values the familiar way, using the standard  $x + yi$  notation. This can be done with the `'__tostring'` method which should return a string.

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

> z:
1+1i
```

Calling the **cmplx** constructor is a bit clumsy, so let us define the imaginary unit  $I = 0+i$  to be used in subsequent operations. Before assigning the imaginary unit, we have to add a metamethod for multiplying a number by a complex number.

```
> cmplx_mt.__mul := proc(a, b) is
>   if a :: 'cmplx' and b :: 'cmplx' then
>     return cmplx(a[1]*b[1]-a[2]*b[2], a[1]*b[2]+a[2]*b[1])
>   elif a :: number and 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 a :: 'cmplx' and b :: 'cmplx' then
>     return cmplx(a[1]+b[1], a[2]+b[2])
>   elif a :: number and b :: 'cmplx' then
>     return cmplx(a+b[1], b[2])
>   fi;
> end;

> i := cmplx(0, 1);

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

Until now, the real and imaginary parts can only be accessed using indexed names, say `z[1]` for the real part and `z[2]` for the imaginary part. A more convenient - albeit not that performant - way is to use a notation like `z.re` and `z.im` in both read and write operations and their respective `'__index'` and `'__writeindex'` metamethods.

This is what will happen if you have defined the `'__index'` metamethod for *reading* values from a structure obj:

- If the structure is a table, then Agena will automatically call the metamethod if the raw lookup `obj[key]` results to **null**.
- If the structure is a set, then Agena will automatically call the metamethod if the raw lookup `obj[key]` results to **false**.
- If the structure is a sequence, register or pair, then the metamethod will be called if the lookup `obj[key]` would result in an index-out-of-range error.

If there is a `'__writeindex'` metamethod for *writing* values to a structure, Agena will do the following:

- If the structure is a table, sequence, register or pair, then the metamethod will always be called first when assigning a value to the expression `obj[key]`.
- The **insert** statement will also call the metamethod.

The procedures assigned to the `'__index'` and `'__writeindex'` keys of a metatable should not include calls to indexed names, for Agena would call the associated metamethods again and again, causing stack overflows. Instead, use the **rawget** function to directly read values from a structure, and the **rawset** function to write values to a structure.

Let us first define a global mapping table for symbolic names to integer keys:

```
> cmplx_indexing := [re ~ 1, im ~ 2];
```

Now let us define the two new metamethods. Both will accept expressions like `a.re` and `a[1]`. In the following read procedure the parameter `x` represents the complex value itself, and parameter `y` the keys `'re'` or `'im'`, or the indices 1 and 2.

Thus, `cmplx_indexing['re']` will evaluate to the index 1, and `cmplx_indexing['im']` to index 2.

```
> cmplx_mt.__index := proc(x, y) is # read operation
>   if y :: string then # for calls like `a.re` or `a.im`
>     return rawget(x, cmplx_indexing[y])
>   else
>     return rawget(x, y) # for calls like `a[1]` or `a[2]`
>   fi
> end;
```

In the write procedure, parameter `x` will hold the complex value, `y` will be either the `'re'` or `'im'` key, or the integer index 1 or 2, and `z` represents the value to be stored, that is `x.re := z` or `x.im := z`.

```
> cmplx_mt.__writeindex := proc(x, y, z) is # write operation
>   if y :: string then
>     rawset(x, cmplx_indexing[y], z)
>   else
>     rawset(x, y, z) # for assignments like `a[1] := value`
>   fi
> end;
```

You can now use the new methods.

```
> a:
1+2i

> a.re:
1

> a.im := 3;

> a:
1+3i
```

Note that while arithmetic metamethods can be applied on mixed types, for example the above defined complex number and a simple Agena number, relational operators cannot *compare* values of different types. Instead, Agena in this case just returns **false** with the equality operators `=`, `==`, and `~=`; and issues an error with relational operators that compare for order.

The **delete** statement supports metamethods, too: it passes the data to be deleted as its key and **null** as the value to the `__writeindex` metamethod. To protect values stored to structures you might define:

```
> readonly_mt.__writeindex := proc(t, k, v) is
>   if unassigned v or assigned rawget(t, k) then
>     error('cannot delete or modify value')
>   else
>     rawset(t, k, v)
>   fi
> end;
```



The **pop**, **rotate**, **duplicate**, and **exchange** statements issue an error if a given structure features a `__writeindex` metamethod. This prevents read-only structures from being modified.

### 6.19.2 Write-Protection

To write-protect a table easily, but also a set, sequence, register, pair or userdata, you may just call the **freeze** function. After execution, you still can read data from the structure, but cannot modify or delete values from it. You can also not read, set or modify its metatable and also not set or change a user-defined type. The write-protection, however, does not prevent a structure from being entirely overwritten or deleted by the **clear** statement or by setting it to **null**.

```
> tbl := [1, 2, 3, a=10, b=20]
> freeze(tbl);
> tbl[1] := null;
Error in stdin at line 1: table is read-only.
```

Use **unfreeze** to remove the protection altogether.

An alternative is the `'__writeindex'` metamethod. In the following example, we will create a procedure that accepts a table, write-protects it and returns it.

The metamethod:

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

A constructor that simplifies creating read-only structures:

```
> readonly := proc(t :: table) is
>   setmetatable(t, readonly_mt);
>   return t
> end;
> moons := readonly(['Phobos', 'Deimos']);
```

Adding further values to the table, or changing an existing one, now will not work.

```
> insert 'Mars' into moons;
Error, structure is read-only.
```

Stack traceback: in ``error``

```
> moons:
[Phobos, Deimos]
```

Using one and the same global table to define metamethods for various variables may be appropriate to save memory, but modification of the metatable itself may have unwanted effects.

```
> readonly_mt.__writeindex := proc(t, k, v) is rawset(t, k, v) end;

> insert 'Mars' into moons;

> moons:
[1 ~ Phobos, 2 ~ Deimos, Mars ~ Mars]
```

Finally, to protect values already assigned to a table, we could define:

```
> readonly_mt := [
>   __writeindex =
>     proc(t, k, v) is
>       if rawget(t, k) <> null then
>         error('Error, structure is read-only.')
>       else
>         rawset(t, k, v)
>       fi
>     end
> ]

> create table t;

> setmetatable(t, readonly_mt)

> t[1] := 0

> t[1] := 1
Error, structure is read-only.
```

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

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

> readonly := proc(t :: table) is
>   setmetatable(t, readonly_mt);
>   return t
> end;

> moons := readonly(['Phobos', 'Deimos']);

> setmetatable(moons, [
>   __writeindex =
>     proc(t, k, v) is error('Error, table is read-only') end
>   ]
> );
Error in `setmetatable`: cannot change a protected metatable.

Stack traceback: in `setmetatable`
  stdin, at line 1 in main chunk
```

Alternatively, you can use the **constant** declaration statement along with **freeze** to completely write-protect a structure or a metatable:

```
> readonly_mt := [ __writeindex = proc() error('Table is read-only') end ];
```

Protect table from tampering:

```
> freeze(readonly_mt);
```

Protect table from being completely overwritten or deleted - this only works in a top-level chunk, though:

```
> constant readonly_mt := readonly_mt;
```

### 6.19.3 Type-Checking

Metamethods can also be used to automatically check the contents of a structure passed at function invocation, and also to extend the `::` and `-:` operators.

Let us assume we would like to write a procedure that sums up all numbers in a set:

```
> s := {1, 2, 3, 4, 5};
```

We create a metatable first,

```
> create table mt;
```

and then assign a proper evaluation procedure to the `__oftype` metamethod that makes sure that the set consists of numbers only.

```
> mt.__oftype := proc(x) is
>   if type x <> set then return false fi;
>   for i in x do # or use sets.isall for short
>     if i -: number then return false fi
>   od;
>   return true
> end
```

We assign the metatable to the set,

```
> setmetatable(s, mt);
```

and first try out the extended `::` and `-:` operators.

```
> s :: set:
true
```

If an invalid member is inserted into the set,

```
> insert 'a' into s;
```

the type check fails:

```
> s :: set:
false
```

```
> s -: set:
true
```

Now we use the type evaluator in a procedure call:

```
> sum := proc(x :: set) is
>   local s := 0; for i in x do inc s, i od; return s
> end;

> sum(s):
In stdin:
  argument #1 does not satisfy type check metamethod
```

The '`__oftype`' metamethod works as follows: it first checks whether the structure (a table, set, sequence, register, pair) or userdata at the left-hand side matches the basic or user-defined type given at the right-hand side. If true, then Agena will check whether the structure has an attached '`__oftype`' metamethod and then will run it. The validator function must either return **true** if the criteria have all been met, or **false**, **fail** or **null** otherwise.

Note that in the validator `mt.__oftype` definition given above, we use the **type** operator instead of the `::` operator in the first **if** statement since otherwise Agena would issue a stack overflow error as it would recursively call the metamethod.

The '`__oftype`' metamethods also work if a return type has been specified.

#### 6.19.4 Calling a Structure like a Function

A structure with a '`__call`' key in its metatable can be called like a function.

```
> readonly := proc(t :: table) is
>   setmetatable(t, [
>     __call = proc(t) is
>       for i, j in t do print(i, j) od
>     end]);
>   return t
> end;

> moons := readonly(['Phobos', 'Deimos']);

> moons();
1      Phobos
2      Deimos
```

#### 6.19.5 Iterating Objects in for/in Loops

Finally, the `__iter` metamethod allows **for/in** to directly loop over userdata and closures. Just put the iterator - not the generating factory - along the `__iter` field of a userdata. numarrays, tuples, singly-linked, doubly-linked and unrolled singly-linked lists automatically have an attached iterator in their metatables.

When you prematurely leave a **for/in** loop that iterates a tuple or userdata and want to iterate it again, you might first want to reset the internal iterator so that it starts from the very beginning of the object, like that:

```
> while getmetatable(object).__iter() do od;
```

### 6.19.6 Registering Metamethods

In some packages, for example `llist` and `numarray`, metamethods are included in the binary C library file and can be accessed through the so-called registry, via the **debug.getregistry** function. You may want to use this function to add further self-defined metamethods written in the Agena language.

For example, the `'__in'` metamethod of the `numarray` package is defined in the Agena source file `lib/numarray.agn`, and not in the C library file.

```
> numarray.aux.mt := [
>   __in = proc(x, a) is
>       return numarray.whereis(x, a, 1, Eps) <> null
>   end
> ]
```

The metatable stored to the registry can be read by a call to **registry.get** or **getmetatable**. Just insert all of your own metamethod procedures by individually adding them, but do not directly assign your metamethod table to the result of **registry.get('numarray')**, **getmetatable('numarray', true)**.

```
> scope
>   # protect against sandboxing (prevent errors at initialisation)
>   if registry.get :: procedure then
>       # get the internal registry metatable for numarrays
>       local _mt := registry.get('numarray');
>       if _mt :: table then
>           # include each metamethod function step-by-step
>           for i, j in numarray.aux.mt do
>               _mt[i] := j
>           od
>       fi
>   fi
> epocs;
```

Never modify or delete existing metamethods, as this will lead to undefined behaviour.

## 6.20 Memory Management, Garbage Collection, and Weak Structures

Agena includes a garbage collector that sweeps all structures, procedures, userdata, and threads (called `objects` in this subchapter) that no longer have valid references in your programme - i.e. are inaccessible. Agena can then use the space for new objects. Numbers, complex numbers, strings and booleans are never collected.

Consider the following code: Let us assign a table to a name.

```
> s := []
```

Now `s` refers to a memory address so that Agena can access the table.

```
> environ.pointer(s):  
008F0F38
```

If we reassign `s`, a different empty table is assigned to it.

```
> s := []
```

This newly created table is stored to another part of the memory.

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

Since the first table at memory position 008F0F38 can no longer be accessed, it unnecessarily occupies space. The garbage collector regularly looks for unreferenced objects and removes them.

Besides automatic garbage collection, the user can also invoke it manually, if deemed necessary, or even stop and restart it by calling **environ.gc**.

Sometimes it may be necessary to immediately clear values occupying a large amount of space. In this case assign **null** to it, so that the next automatic collection cycle can free it. If necessary call **environ.gc** for immediate collection. As a shortcut, you could also use the **clear** statement which conducts both **nulling** a value and collecting it.

If a table, set, sequence, procedure, userdata or thread is included in another table or sequence, the garbage collector does not collect it if its reference should have become invalid.

```
> restart  
  
> t := []  
  
> v := [1]; insert v into t  
  
> v := [2]; insert v into t
```

```
> environ.gc()
```

[1] is still part of the table.

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

If you do not want this to happen, declare the table or sequence ``weak`` by using the `'__weak'` metamethod. With tables, you can either declare its keys weak by passing the string `'k'`, or its values weak with the string `'v'`, or both with `'kv'`. With sequences, simply use the string `'v'`.

If the collector meets a weak key that has become inaccessible, it removes the key-value pair. If the collector meets a weak value that has become inaccessible, it removes the key-value pair.

```
> t := []

> setmetatable(t, ['__weak' ~ 'v'])

> v := [1]; insert v into t

> v := [2]; insert v into t

> environ.gc()

> t:
[2 ~ [2]]
```

Do not change the `'__weak'` field after it has been assigned to an object, as the behaviour would be undefined. The **insert** and **delete** statements will reject manipulation of weak tables and sequences.

## 6.21 Extending Built-in Functions

You may redefine existing built-in functions if you want to change their behaviour or extend its features. You can either write a completely new replacement from scratch or use the original function in your modified version. Your new procedure can then be called with the same name as the original one.

Note that only Agena functions written in C or in the language itself can be redefined, and that operators cannot.

In Agena, each mathematical function  $f$  works as follows: if a number  $x$ , which by definition represents a value in the real domain, is passed to them, then the result  $f(x)$  will also be in the real domain. If  $x$  is a complex value, then the result will be in the complex domain.

Suppose that you want to automatically switch to the complex domain if a function value in the real domain could not be determined, i.e. if  $f(x) = \text{undefined}$ . An example is:

```
> root(-2, 2):
undefined
```

On the interactive level enclose the new procedure definition with the **scope** and **epocs** or **begin/end** keywords. This is necessary because on the interactive level, each statement entered at the prompt has its own scope and thus local variables cannot be accessed in the statements thereafter.

The new function definition might be:

```
> begin
>   # save the original function in a `hidden` variable
>   local oldroot := root;
>
>   # define the substitute
>   root := proc(x, n) is # new definition
>     local result := oldroot(x, n);
>     if result = undefined then # switch to complex domain
>       result := oldroot(x+0*I, n)
>     fi;
>     return result
>   end;
>
> end;
```

The original function `root` is stored to the local `oldroot` variable so that the user can no longer directly access it.

```
> root(-2, 2):
8.6592745707194e-017+1.4142135623731*I
```

If you wish to permanently use your redefined functions, just put them into the initialisation file, located either in the `lib` folder of your Agenda installation, or your home directory. See Appendix 6 for further information.

## 6.22 Closures: Procedures that Remember their State

A procedure can remember its state. This state is represented by the function's local variables which survive and retain their values even after the call to the procedure has finished. Such procedures are also called `closures`.

So with successive calls, the procedure can access these values again and re-use them.

Let us define an iterator function that returns an element of a table one after the other:

```
> traverse := proc(o :: table) is
>   local count := 0;
>   return proc() is
>     inc count;
>     return o[count]
>   end
> end;
```



The `traverse` procedure is called a `factory` as it creates and returns the closure which we assign to the name `iterator` subsequently:

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

The `iterator` function remembers its state and can be called like `normal` functions:

```
> iterator():
a
```

What happened ? The call to `traverse` with the table `tbl = ['a', 'b', 'c']` as its only argument initialised the variable `count` and assigned it to 0. The table you passed is also stored to the closure's internal state since technically, parameters are local variables. With the first call to `iterate`, `count` was incremented from 0 to 1, so that the first element of the table, i.e. `tbl[1]`, could be returned thereafter.

```
> iterator():
b
> iterator():
c
```

Since the table now has no more elements left (`count = 4`), the iterator now returns **null**, since `tbl[4] = null`.

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

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

```
> iterator2 := traverse(['a', 'b', 'c']);
> iterator2():
a
> iterator2():
b
> iterator3 := traverse(['a', 'b', 'c']);
> iterator3():
a
```

In Chapter 5, we have already introduced **for/in** loops that can iterate over functions. There are various ways to accomplish this.

In general, one or two loop control variables are given to the left of the **in** keyword, followed by the function and up to two further variables to its right.

Example 1: With function **nextone** iterate table `tbl` and pass **null** as the initialiser to get its first entry. The respective values in `tbl` are assigned to loop control variable `i`:

```
> tbl := [10, 20, 30, 'a' ~ 40];

> for i in nextone, tbl, null do # equivalent to `for i in tbl do`
>   print(i)
> od;
10
20
30
40
```

Example 2: Same as Example 1 but with two control variables `k`, `v` storing the respective table key and value, in this order.

```
> for k, v in nextone, tbl, null do
>   print(k, v)
> od;
1      10
2      20
3      30
a      40
```

Example 3: Retrieve only the table keys.

```
> for keys k in nextone, tbl, null do
>   print(k)
> od;
1
2
3
a
```

**for/in** loops iterate over factories, as well. Just some examples:

```
> gmatch := proc(s) is
>   local c, p := 0, strings.gmatch(s, '%a+'); # p is assigned an iterator
>   return proc() is
>     local word := p();
>     return when word = null with null;
>     inc c;
>     return c, word # return position and word
>   end
> end;

> s := 'hello world from Agena';

> f := gmatch(s);

> for i in f do
>   print(i)
> od;
hello
world
from
Agena

> f := gmatch(s);
```

```
> for k, v in f do
>   print(k, v)
> od;
1      hello
2      world
3      from
4      Agena

> f := gmatch(s);

> for keys k in f do
>   print(k)
> od;
1
2
3
4
```

## 6.23 Self-defined Binary Operators

A procedure `f` of two arguments `x, y`

```
> plus := proc(x, y) is return x + y end;
```

can be called like a binary operator through the syntax `x f y`:

```
> 1 plus 2:
3
```

When using a function as a binary operator, it has always the highest precedence.

## 6.24 OOP-style Methods on Tables

Agena supports OOP-style methods. For a table object representing a bank account,

```
> account := ['balance' ~ 0];
```

define the following method (please note the two `@` tokens):

```
> proc account@@deposit(x) is
>   inc self.balance, x;
>   return self.balance
> end;
```

The name `self` always refers to the table object, here `account`. Call the method using two `@` characters:

```
> account@@deposit(100)
```

Query the object.

```
> account:
[balance ~ 100, deposit ~ procedure(016D6820)]
```

Let us define a method for withdrawing an amount of money. Instead of the **proc** statement, we will now use the standard `:=` assignment:

```
> account@@withdraw := proc(x) is
>   if x < 0 then error('Error, value must be non-negative.') fi;
>   dec self.balance, x;
>   return self.balance
> end;
```

To set up new accounts that inherit the methods and characteristics associated with the `account` object, assign the metatable of the `account` object to the freshly created account using the **setmetatable** function, and force Agena to search for the methods or its balance stored to `account` by proper indexing (i.e. `self.__index := self`). Thus, we use the `account` object as a prototype inherited by individual accounts. To explore the metatable of an object, call **getmetatable**.

```
> proc account@@new(o) is
>   o := o or [];          # if not given, create object with its initial
>                           # balance taken from the current state of `account`
>   setmetatable(o, self); # assign metatable of `account` object
>                           # (i.e. `self`) to new table
>   self.__index := self;  # inherit methods from `account` object
>   return o
> end;

> a := account@@new();

> a.balance:
100
```

Set up a new account with its initial balance set to zero:

```
> b := account@@new(['balance' ~ 0]);
```

Pay into the bank 200 currency units.

```
> b@@deposit(200):
200
```

If you want to create a different class of accounts, e.g. accounts on credit that own all the features of `account` but do not allow any overdraft, just assign an `account` object to it by calling the `new` method (do not just assign `account` to `creditaccount`):

```
> creditaccount := account@@new();
```

and overwrite the `withdraw` method:

```
> proc creditaccount@@withdraw(x :: number) is
>   if x < 0 then error('Error, value must be non-negative.') fi;
>   if x > self.balance then error('Error, not enough credit.') fi;
>   dec self.balance, x;
>   return self.balance
> end;
```

```
> c := creditaccount@@new();

> c@@withdraw(1000):
Error, not enough credit.
```

Since `b` is an unlimited account, we can withdraw money as much as we want, as its `withdraw` metamethod has not been replaced.

```
> b@@withdraw(1000):
-800
```

## 6.25 Assigning Tables to Procedures

As an alternative to storing values into the registry (see Chapter 6.31) or using closures (Chapter 6.22), you can assign a table to a procedure with the **store** feature.

This table will remain active during the entire Agenda session and you can read from or write values to it in subsequent calls to the function.

This feature is thrice as fast as interacting with the registry, but only half as fast as closures. The table can be accessed through the **store** keyword which can also be indexed:

```
> f := proc() is
>   feature store;
>   store[1] := Pi;
>   store.count := (store.count or 0) + 1;
>   return store, store[1], store.count
> end;

> f()
[3.1415926535898, 1] 3.1415926535898 1
```

To get access to the internal store, call **debug.getstore** which returns its reference. You can both inspect this table as well as inject values into the store. In the following example we define a sine function with precomputed coefficients:

```
> zxssine := proc(x :: number) is # ZX Spectrum SIN emulation
>   feature store; # activate the internal store
>   local w, z;
>   x := 0.5/Pi;
>   x := entier(x + 0.5);
>   w := 4*x;
>   if w > 1 then
>     w := 2 - w
>   elif w < -1 then
>     w := -w - 2
>   fi;
>   z := 2*w*w - 1;
>   return w*(store[1] + z*(store[2] + z*(store[3] + z*(store[4] +
>     z*(store[5] + z*store[6])))))
> end;

> _coeffs := // 1.267162131 -0.284851843 0.18226552e-1 -0.546208e-3
>             0.9480e-5 -0.112e-6 \\\;
```

To get a reference to the store, execute:

```
> _store := debug.getstore(zxsine);
```

Insert coefficients into the store,

```
> for i in _coeffs do  
>   insert i into _store  
> od;
```

and do some cleanup thereafter:

```
> _store, _coeffs -> null;
```

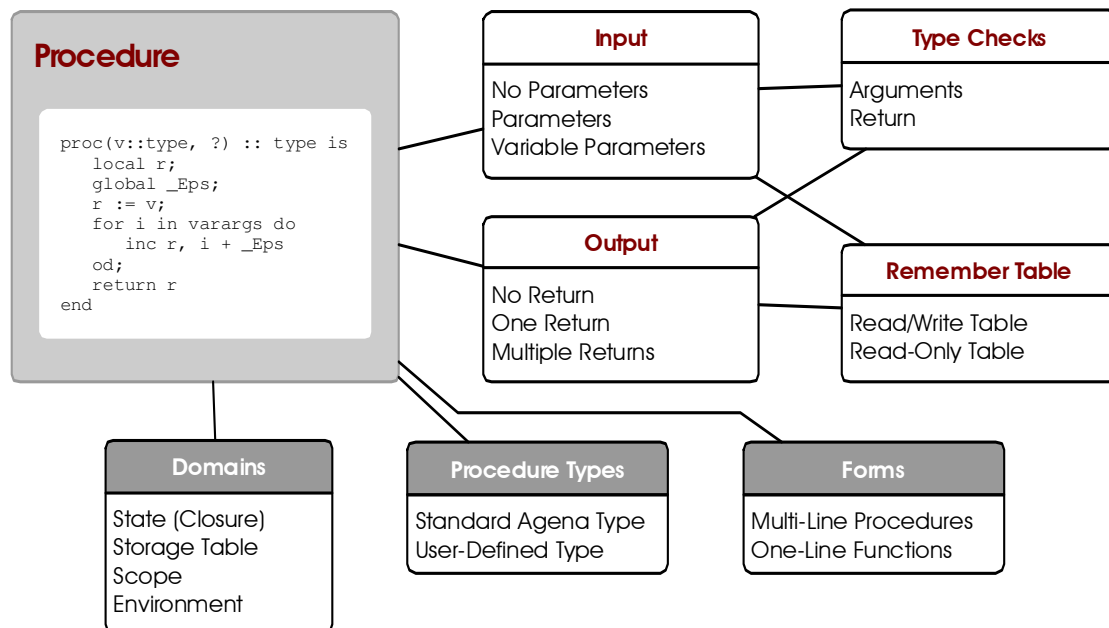
Voilà:

```
> zxsine(Pi/4):  
0.70710678125
```

Of course, you can mix **store** tables with remember tables. For another example, see Chapter 6.31.

## 6.26 Summary on Procedures

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



## 6.27 I/O

Agena features various functions to deal with files, to read lines and write values to them. Keyboard interaction is supported, too, as is interaction with other applications. Most of the functions have been taken from Lua. All the functions for input/output are included in the **io** and the **binio** packages.

Read and write access to files usually is conducted through file handles. At first, a file is opened for read or write operations with the **io.open** function. Then you apply the respective read or write functions and finally close the file again by calling **io.close**.

### 6.27.1 Reading Text Files

Open a file and store the file handle to the name `fh`:

```
> fh := io.open('d:/agenda/src/change.log'):
file(7803A6F0)
```

Read the first ten characters:

```
> io.read(fh, 10):
Change Log
```

Read the next ten characters:

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

Close the file:

```
> io.close(fh):
true
```

Besides file handles, many I/O functions also accept file names. For example, the **io.lines** procedure reads in a text file line by line. It is usually used in **for** loops. The respective line read is stored to the loop key, the loop value is always **null**. The function opens and closes the file automatically.

```
> for i, j in io.lines('d:/agena/lib/agena.ini') do
>   print(i, j)
> od
execute := os.execute;      null
getmeta  := getmetatable;   null
setmeta  := setmetatable;   null
```

### 6.27.2 Writing Text Files

To write numbers or strings into a file, we must first create the file with the **io.open** function. The second argument `'w'` tells Agena to open it in `'write'` mode.

```
> fh := io.open('d:/file.txt', 'w');
```

As mentioned above, **io.open** returns a file handle to be used in subsequent I/O operations.

```
> io.write(fh, 'I am a text.');
```

If you would like to include a newline, pass the `'\n'` string,

```
> io.write(fh, 'Me ', 'too.', '\n');
```

or use the **io.writeline** function which automatically adds a newline to the end of the input. The next statement writes the number  $\pi$  to the file.

```
> io.writeline(fh, Pi);
```

After all values have been written, the file must be closed with **io.close**.

```
> io.close(fh);
```

The statements presented above produce the file contents:

```
I am a text.Me too.
3.1415926535898
```



We can append text to a file we have already created. In order to append - and not to overwrite existing - text, use the 'a' switch in the call to **io.open**<sup>16</sup>. Using the 'w' switch would replace the text already existing with the new one. See Chapter 12.1 for further options accepted by **io.open**.

Tables, sets or sequences cannot be written directly to files, they must be iterated using loops so that their keys and values - which must be numbers, booleans or strings - can be stored separately to the file thereafter. The same applies to pairs: use the **left** and **right** operators to write their components.

The following statements write all keys and values of a table to a file. The keys and values are separated by a pipe '|', and a newline is inserted right after each key~value pair. Note that you can mix numbers and strings.

```
> a := [10, 20, 30];
> file := io.open('d:/table.text', 'w');
> for i, j in a do
>   io.write(file, i, '|', j, '\n')
> od;
> io.close(file);
```

Hint: To create UNIX text files on DOS-like systems, such as DOS, OS/2, Windows, just open the text file in binary mode, e.g. `io.open('d:/table.text', 'wb')`. This avoids carriage return control codes to be added to the file with each line break.

See Chapter 12.1 for a description of all **io** package functions.

If you have trouble with character encoding, the converters **strings.tolatin**, **strings.toutf8**, **strings.diamap** or the **iconv** package might help you. You may also use **tostringx** to easily write any data to a text file and read it back later.

### 6.27.3 Keyboard Interaction

The **io.read** function allows to enter values interactively via the keyboard when called with no argument. Use the RETURN key to complete the input. The value returned by **io.read** is a string. If you would like to enter and process numbers thereafter, use the **tonumber** function to transform the string into a number.

```
> a := io.read();
10
> a:
10
> type(a):
string
> tonumber(a)^2:
100
```

---

<sup>16</sup> See Chapter 12.1 for further options accepted by **io.open**.

All available keyboard functions are:

Procedure	Details
<b>io.anykey</b>	Checks whether a key has been pressed and returns <b>true</b> or <b>false</b> .
<b>io.getkey</b>	Waits until a key is pressed and returns its ASCII value. This function is not available for all platforms.
<b>io.read</b>	If called with no arguments, reads one or more characters from the keyboard until the RETURN key is being pressed. The return is a string.

Table 19: Functions to read the keyboard

#### 6.27.4 Default Input, Output, and Error Streams

Agena provides aliases to the standard input, output, and error channels known from C:

- **io.stdin**, the standard input stream, used to input data, usually the keyboard,
- **io.stdout**, the standard output stream, used to output data, usually the console,
- **io.stderr**, the standard error stream, used for error messages and diagnostics, usually the console.

Examples:

```
> io.writeline(io.stdout, 'Okay');
Okay

> io.writeline(io.stderr, 'Not okay');
not okay
```

#### 6.27.5 Locking Files

Agena allows files to be locked so that only the current process can read or write data to them. This feature prevents corruption to files during write operations or reading invalid data when other programmes try to access them. See **io.lock** and **io.unlock** in Chapter 12.1 for further information.

#### 6.27.6 Interaction with Applications

You can call another application, pass data to it and receive data from the application with the **io.popen** function. The function returns a file handle, so that you can receive the information returned (from the stdout channel of the called programme) for further processing.

To get a listing of all files in the current directory, enter:

```
> p := io.popen('ls'):
file(77602960)

> io.readlines(p):
[ads.c, agena.c, etc.]
```

Finally, close the connection.

```
> io.close(p)
```

If you pass the 'w' option to **io.popen** as a second argument, you can send further data to the external programme:

```
> p := io.popen('cat', 'w')
> io.write(p, 'Hello ')
> io.write(p, 'World\n')
> io.close(p)
Hello World
```

If you want to receive data from the stderr channel, or suppress output at the Agenda console, include the respective redirection instruction, which may vary among operating systems, in the first argument to **io.popen**.

### 6.26.7 CSV Files

Comma-separated value files can be read and written conveniently by **utils.readcsv** and **utils.writecsv**. This function provides various options to further process the data being read. See Chapter 16.1 for further details.

### 6.27.8 XML Files & JSON Objects

XML files are imported and converted to Agenda data structures with **utils.readxml** or **xml.readxml**. XML files can be created with **utils.encodexml** and **io.write**. Chapter 16.1 and 12.5 offers further information on how to do this.

JSON objects represented by strings can be converted to Agenda tables with **json.decode** and written from a table to a 'JSON string' with **json.encode**.

### 6.27.9 dBASE III/IV & SQLite Files

The **xbase** package can read and write dBASE III/IV-compatible files. See Chapter 12.3 for details.

Likewise, the **sqlite** package interfaces to SQLite databases, see file 'SQLite.htm' in the 'doc' folder of your Agenda installation.

### 6.27.10 INI Files

The **utils.readini** and **utils.writeini** functions as well as the **ini** package deal with traditional INI initialisation files.

## 6.28 Linked Lists

With large tables, sometimes it may be very costly to insert or delete an element with the **put** and **purge** functions because all elements after the insert or deletion position must either be shifted up- or downwards. This is also true with sequences and registers.

In addition, iterating a table with the **for/in** statement does not ensure that the keys are traversed in ascending order<sup>17</sup>.

In these cases you may use the **llist** package implementing linked lists which store elements in a sequential order and where each value also links to its successor (and predecessor). Just take a look at the examples at the end of this subchapter.

The benefit of using linked list in these situations is a speed increase of at least 600 %, but may be very much larger.

To see how a linked list works, let us create one manually. First, establish a root which indicates the end of the list.

```
> list := null;
```

Now we insert the numbers -2, -1 and 0 into this list, so that it contains the elements 0, -1, -2, in this order.

```
> list := ['data' ~ -2, 'nextone' ~ list];
```

```
> list := ['data' ~ -1, 'nextone' ~ list];
```

```
> list := ['data' ~ 0, 'nextone' ~ list];
```

To traverse the list, we use a new reference so that the original list is not changed:

```
> l := list;
```

```
> while l do
>   print(l.data)
>   l := l.nextone
> od;
0
-1
-2
```

To insert an element somewhere in the list, we enter:

```
> l := list;
```

```
> while l do
>   if l.data = -1 then
>     l.nextone := ['data' ~ -1.5, 'nextone' ~ l.nextone];
>     break
>   fi;
>   l := l.nextone
```

---

<sup>17</sup> See **skycrane.iterate**.

```
> od;
> l := list;

> while l do
>   print(l.data)
>   l := l.nextone
> od;

0
-1
-1.5
-2
```

It may often be useful to add further information to a linked list to save unnecessary traversal, e.g. the position of the element or the predecessor.

Instead of implementing singly or doubly-linked lists yourself, use the **llist** package.

Create an empty list.

```
> L := llist.new():
llist()
```

Now add 0 to it

```
> llist.append(L, 0);
```

and also put -2 to its beginning.

```
> llist.prepend(L, -2);

> L:
llist(-2, 0)
```

Insert -1 at position 2. As you see, the original element at this position is not deleted but shifted to open space.

```
> llist.put(L, 2, -1):

> L:
llist(-2, -1, 0)
```

To delete an element at a position, enter:

```
> llist.purge(L, 2):

> L:
llist(-2, 0)
```

The **size** operator determines the number of all elements in a linked list.

```
> size L:
2
```

To determine a specific element, index it as usual:

```
> L[1]:
-2
```

Passing an index that does not exist, simply results to **null**.

Finally, to replace an element, use a usual assignment statement.

```
> L[2] := -1

> L:
l1ist(-2, -1)
```

You may have a look at unrolled singly-linked lists, which are also provided by the **l1ist** package for high-speed processing. The **ul1ist** functions have the same name as those for **l1ists**, and almost the same syntax, so here is just a small example:

```
> a := ul1ist.new(64)      # 64 slots per node

> for i to 11 do ul1ist.append(a, i) od  # fill ul1ist with numbers 1 to 11

> ul1ist.put(a, 5, 100);  # insert 100 at position 5

> a := ul1ist.dump(a);    # convert ul1ist into a sequence and dump it
>                               # from memory
> print(a)
seq(1, 2, 3, 4, 100, 6, 7, 8, 9, 10, 11)
```

Finally, functions to work on doubly-linked lists are available in table **d1ist**. Read and write access to elements in doubly-linked lists is around twice as fast as for singly-linked lists:

```
> l := d1ist.new('Algol 68', 'Maple', 'Lua', 'SQL');

> d1ist.append(l, 'Agena');  # add new entry to the end of the list

> l[-1]:
Agena

> d1ist.prepend(l, 'Agena'); # add new entry to the start of the list

> l[1]:
Agena

> d1ist.purge(l, 1);         # delete first entry

> l[1]:
Algol 68

> d1ist.purge(l, -1);       # delete last entry

> l[-1]:
SQL

> # insert a new value into the middle of the list, shifting elements into
> # open space
> d1ist.put(l, 3, 'Agena');
```

```
> dlist.toseq(l):
seq(Algol 68, Maple, Agena, Lua, SQL)

> f := dlist.iterate(l); # iterate through the list

> f():
Algol 68
(etc.)
```

## 6.29 Numeric C Arrays

Agena numbers can alternatively be processed using numeric C arrays. The `numarray` package supports C doubles, signed 4-byte integers (`int32_t`), and unsigned chars. See Chapter 10.6 for further details.

While C numeric arrays consume less memory than Agena's built-in structures, operations are slower.

## 6.30 Userdata and Lightuserdata

Some Agena packages such as linked lists and `numarrays` implement data structures by so-called `userdata`, i.e. C structures that are garbage-collected by the interpreter provided that a `'__gc'` metamethod exists.

Likewise, `lightuserdata` are pointers to any C objects but programmers writing C libraries have to implement their own garbage collection procedures.

To the ordinary programmer writing code exclusively in the Agena language, `userdata` and `lightuserdata` are irrelevant as this kind of data can only be accessed through functions written in C.

If you are looking for an empty `userdata` structure with an accompanying data container that you can programme on the Agena level, you might try **`utils.udata`**.

## 6.31 The Registry

The registry is an interface between Agena and its C virtual machine which mainly stores values needed by `userdata`, metatables of libraries written in C, open files, and loaded libraries. It can also be used to exchange data between the C environment and Agena, or between Agena functions in general. See Chapter 6.25 for a faster alternative if you know that a function does not need to exchange data with other functions.

**`debug.getregistry`** gives full access to the registry but should be used carefully. It is recommended to revert to the functions of the **`registry`** package to read, add or delete registry data or to modify C library metatables, and to exclude the **`debug`** library from sandboxes (see Chapters 6.15, 7.40 and 7.53).

Registry entries indexed by integral keys refer to data occupied by userdata objects, which for example are used by the **llist** and **numarray** libraries. The **registry** library, however, does not expose these values to Agena.

Following is an example how you can use this feature:

```
> watch := proc(x) is
>   local id, t, val;
>   t := time();
>   # create light userdata as registry key
>   id := 'baselib_watch';
>   unassigned registry.get(id) ? registry.anchor(id, 0);
>   if x then # any argument given ? -> initialise / reset the clock
>     registry.anchor(id, 0);
>     return
>   fi;
>   val := registry.get(id); # get old time (in seconds)
>   if val = 0 then # start clock
>     registry.anchor(id, t); # assign a new value to registry
>     t := 0
>   else # return elapsed time and set clock to current time
>     t -= val;
>     registry.anchor(id, time())
>   fi;
>   return t
> end;
```

In comparison, an implementation using an internal store table would be:

```
> watch := proc(x) is
>   feature store;
>   local id, t, val;
>   val := store[1]; # get old time (in seconds)
>   unassigned val ? store[1] := 0; # initialise with the first call
>   t := time();
>   if x then # reset the clock but do not turn it on again
>     store[1] := 0;
>     return
>   fi;
>   if val = 0 then # start clock
>     store[1] := t;
>     t := 0
>   else # return elapsed time and set clock to current time
>     t -= val;
>     store[1] := time()
>   fi;
>   return t
> end;
```



## 6.32 Functional-Style Programming

Agena features operators and functions that spare you some lines of code by applying a function on a structure or numeric range, for example to transform values, to select or remove values, accumulate values, etc.

There are also iterators, and functions that try out a bunch of functions on objects, zip together structures, create composite functions or transform functions with multiple arguments into sequences of single-argument functions. All this allows for some sort of pseudo functional-style programming.

Many of the functions and operators do not only work with structures, but with strings, as well.

**map** applies a function on each element of a structure. To square all numbers in a table, enter:

```
> map(<: x -> x^2 :>, [1, 2, 3]):
[1, 4, 9]
```

**map** works on both univariate and multivariate functions. It either returns a new function or works in-place. It can also reorder elements.

```
> map(<: x, y -> x^2 + y^2 :>, [1, 2, 3], 2): # = x^2 + 2^2 = x^2 + 4
[5, 8, 13]
```

The `inplace` option works destructively and changes the input structure, saving memory especially with large structures:

```
> a := [1, 2, 3];

> map(<: x -> x^2 :>, a, inplace = true):
[1, 4, 9]

> a:
[1, 4, 9]
```

The **@** operator works like **map**, but with univariate functions only:

```
> <: x -> x^2 :> @ a:
[1, 16, 81]
```

The **@** operator allows to create composite functions:

```
> f := <: x -> x^2 :>

> g := <: x -> x + 1 :>

> (f@g)(2):
9

> f(g(2)):
9
```

**select** picks all the elements in a structure that satisfy a Boolean condition. To retrieve all even numbers, type:

```
> select(<: x -> even x :>, [0, 1, 2, 3, 4]):
[1 ~ 0, 3 ~ 2, 5 ~ 4]
```

We can also reorder the elements consecutively with the `reshuffle` option:

```
> select(<: x -> even x :>, [0, 1, 2, 3, 4], reshuffle=true):
[0, 2, 4]
```

**select** can work destructively, in-place:

```
> a := [0, 1, 2, 3, 4];

> select(<: x -> even x :>, a, inplace = true):
[1 ~ 0, 3 ~ 2, 5 ~ 4]

> a:
[1 ~ 0, 3 ~ 2, 5 ~ 4]
```

With univariate functions, the **\$** operator works like **select**. Contrary to **select**, it automatically reorders the elements:

```
> <: x -> even x :> $ [0, 1, 2, 3, 4]:
[0, 2, 4]
```

The **\$\$** operator tests whether at least one element in a structure satisfies a Boolean condition:

```
> <: x -> x < 3 :> $$ [0, 1, 2, 3, 4]:
true

> <: x -> x > 4 :> $$ [0, 1, 2, 3, 4]:
false
```

The **\$\$\$** operator counts the number of items that satisfy a Boolean condition:

```
> <: x -> x < 3 :> $$$ [0, 1, 2, 3, 4]:
3
```

**remove** deletes all the elements satisfying a condition from a structure. In default mode, it returns a new structure leaving the input structure unchanged. The `inplace = true` option, however, actually removes all the values from the input.

```
> a := [0, 1, 2, 3, 4]:
[0, 1, 2, 3, 4]

> remove(<: x -> x < 3 :>, a, inplace = true):
[4 ~ 3, 5 ~ 4]
```

**selectremove** combines the functionality of both **select** and **remove**:

```
> selectremove(<: x -> x < 3 :>, [0, 1, 2, 3, 4]):
[0, 1, 2]          [4 ~ 3, 5 ~ 4]
```

**zip** combines two structures of equal size:

```
> zip(<: x, y -> x + y :>, [0, 1, 2], [10, 11, 12]):
[10, 12, 14]
```

**sumup** approximates series:

```
> sumup(<: n -> 1/fact(n) :>, 0, 15):
2.718281828459
```

```
> sumup(<: n -> 1/fact(n) :>, [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]):
2.7182818011464
```

**mulup** computes products:

```
> 2*mulup(<: n -> 4*n^2/(4*n^2 - 1) :>, 1, 5000): # = Pi
3.1414355935899
```

**foreach** generates structures:

```
> foreach(0, 1, 0.1, <: x -> x :>):
[0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1]
```

It can also insert elements into existing structures which may be empty or non-empty:

```
> s := [-0.1];

> foreach(0, 1, 0.1, <: x -> x :>, s);

> s:
[-0.1, 0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1]
```

The **times** operator takes a start value, applies a function to it and repeatedly applies it to its previous result. To compute the Golden ratio, type:

```
> times(<: x -> 1 + recip x :>, 1, 33):
1.6180339887499
```

**pipeline** maps one or more functions on a structure, from left to right - in the following example we first square each element and then add 1:

```
> pipeline(<: x -> x^2 :>, <: x -> x + 1 :>, [1, 2, 3]):
[2, 5, 10]
```

**reduce** applies a function on each item of a structure or string and returns an accumulated - that is summed-up - result. In the following example the second parameter *a* is the accumulator, by default initialised to zero.

```
> reduce(<: x, a -> x + a :>, [1, 2, 3, 4]):
10
```

By passing a third argument, you can set the initialiser:

```
> reduce(<: x, a -> x + a :>, [1, 2, 3, 4], 10):
20
```

With the `fold` option, the very first element in the structure initialises the accumulator.

```
> reduce(<: x, a -> x + a :>, [10, 1, 2, 3, 4], fold = true):
20
```

**fold** is a shortcut:

```
> fold(<: x, a -> x + a :>, [10, 1, 2, 3, 4]):
20
```

**factory.count** returns an iterator that counts up or down. When the stop value is exceeded, it returns **null**. The function tries to minimise round-off errors as much as possible.

```
> f := factory.count(0, 0.1, 1);

> f():
0

> f():
0.1

...

> f():
0.9

> f():
1

> f():
null
```

Chapter 16.3 `factory - Iterators` explains further iterating functions and some other functional-style procedures, for example **factory.curry** which transforms a function with multiple arguments into a sequence of single-argument functions:

```
> f := <: x, y, z -> x*y + z :>

> t := curry(f); # returns f

> t(10, 20, 30):
230

> t := curry(f, 10); # returns f(10, y, z)

> t(20, 30):
230
```

```
> u := curry(t, 20); # returns t(10, 20)(z) = f(10, 20, z)
> u(30):
230
```



## Index





## A

AgenaEdit, 53  
 Algol 68, 35  
 ArcaOS, 47  
 Arithmetic, 56, 77, 79
 

- Addition, 80
- Arbitrary Precision, 79
- Bitwise Operators, 82, 84
- Complex Math, 85, 86, 87, 89, 92, 93, 95
- Conditional Multiplication, 80, 83
- dec Statement, 83
- Degrees, 78
- div Statement, 83
- Division, 80
- Exponentiation, 77, 80
- Higher & Lower Bits, 82
- inc Statement, 83
- Increment and Decrement, 82, 84
- Integer Division, 80
- Kahan-Babuška Summation, 154
- Kahan-Ozawa summation, 154
- Modulo, 80
- mul Statement, 83
- Multiplication, 80
- Neumaier Summation, 154
- Operators, 80
- Operators & Functions, Overview, 80
- Percentage, 77, 78, 80
- Powers, 80
- Radians, 78
- Subtraction, 80

 Arrays, 107  
 Assignment, 56, 60, 61, 73, 112, 121, 125
 

- Compound Assignment, 83
- Constants, 74, 84, 174
- Defining new Variables, 197
- Enumeration, 75
- Multiple Assignment, 73, 75
- Mutate Operators, 83
- Short-Cut Multiple Assignment, 74
- Unassignment, 75

 Assumptions, 183

## B

Block, 192  
 Booleans, 58, 71, 105, 179

Expressions, 104  
 fail, 104, 105  
 Logical Operators, 104  
 Relational Operators, 104  
 Short-Circuit Evaluation, 105

## C

Calculus
 

- Products, 155
- Series, 155

 Cantor Sets
 

- (please see Sets), 120

 Captures, 98  
 case Statement, 61, 146, 147, 148
 

- Fall Through, 147
- of Clause, 147
- onsuccess Clause, 147
- then Clause, 147

 clear Statement, 56, 75, 214  
 cls Statement, 55  
 Comments, 65  
 Complex Numbers, 57, 71, 86
 

- Cartesian Notation, 57, 85
- Operators, 85
- Polar Notation, 57, 85

 Conditions, 61, 141
 

- case Statement, 146, 147, 148
- Evaluation Rules, 141, 144, 149
- if Operator, 144, 145
- if Statement, 141

 Configuration
 

- Prompt, 55

 Console, 45, 47, 53, 65, 106, 204
 

- cls Statement, 55
- Command Line Usage, 66
- restart Statement, 55

 Constants
 

- fail, 105
- false, 104
- Golden Ratio, 186, 235
- null, 105
- true, 104

 Counting
 

- in Structures, 234

 create Statement, 109, 111, 113, 125, 130, 135  
 CSV Files, 227

**D**

## Data Types

- Boolean, 104
- Complex Numbers, 85
- Lightuserdata, 138, 231
- Number, 77
- Pair, 130
- Register, 138
- Sequence, 123
- Set, 120
- String, 92
- Table, 106, 111
- Thread, 138
- Userdata, 138, 231
- User-defined, 124, 131, 191, 205

## Database

- dBASE III/IV, 227
- SQLite, 227

## Date &amp; Time

- Introduction, 89

## dBASE Files, 227

## dec Statement, 83

## delete Statement, 110, 126, 136

## Dictionaries, 111

## do/as Loops, 62, 151

## do/od Loops, 151

## DOS, 47, 53

**E**

## enum Statement, 75

## Environment

- Reading the Environment of a Procedure, 195
- See also `System Variables/\_G`, 194
- Setting an Environment for a Procedure, 194

## Errors

- Catching Errors, 184, 185, 186
- Issuing Errors, 180
- try/catch Statement, 185, 186

## Escape Sequences, 94

**F**

## Files

- CSV Files, 227
- UNIX Text Files, 225

## for/as Loops, 161

## for/downto Loops, 155

## for/in Loops, 155

## for/to Loops, 62, 153

## for/until Loops, 161

## for/while Loops, 63, 160

## Functional Programming

- \$ Operator, 117, 122, 234
- \$\$ Operator, 122, 234
- \$\$\$ Operator, 234
- @ Operator, 117, 122, 233
- addup Operator, 235
- Currying, 236
- factory.count, 236
- factory.curry, 236
- fold, 236
- foreach Operator, 235
- map, 116, 233
- mulup Operator, 235
- pipeline, 235
- reduce, 235
- remove, 117, 234
- select, 117, 234
- selectremove, 235
- times Operator, 235
- zip, 118, 235

## Functions &amp; Operators

- , 77, 85
- !, 77, 85
- \$, 77, 115
- \$\$, 77, 115
- %, 77, 80
- &, 77
- &&, 77, 82
- \*, 77, 85
- \*\*, 77, 80, 85
- /, 85
- :, 130
- :-, 77, 124, 179
- ::, 77, 124, 179
- @, 115, 117
- \, 77
- ^, 77, 85
- ^ ^, 77, 82
- |, 77
- |-, 77
- ||, 82
- ~~, 77, 82
- ~<>, 77
- ~=, 77
- +, 77, 85
- <, 77, 86, 104

<=, 77, 86, 104  
 <>, 77, 85, 86, 104, 113, 122, 126, 132, 135  
 =, 77, 85, 86, 104, 113, 122, 126, 132, 135  
 ==, 77, 104, 113, 122, 126, 132, 135  
 >, 77, 86, 104  
 >=, 77, 86, 104  
 abs, 85, 96  
 and, 77, 104  
 arccos, 85  
 arcsin, 85  
 arctan, 85  
 assume, 183  
 atendof, 77, 95, 97  
 bottom, 127, 136  
 char, 96  
 Composite Functions, 233  
 copy, 114, 122, 127, 136  
 cos, 85  
 cosh, 85  
 debug.getregistry, 213, 231  
 dec, 77, 87  
 div, 77, 87  
 entier, 85  
 environ.getffenv, 195  
 environ.globals, 174  
 environ.kernel, 82, 106  
 environ.setffenv, 194  
 exp, 85  
 filled, 113, 122, 127, 136  
 finite, 174  
 getentry, 108, 127, 136  
 getmetatable, 128, 132, 137  
 gettype, 124, 128, 131, 132  
 in, 77, 95, 97, 104, 113, 122, 127, 132, 136  
 inc, 77, 87  
 instr, 97, 100  
 intdiv, 77, 87  
 intersect, 77, 114, 122, 128, 137  
 io.anykey, 226  
 io.close, 224, 227  
 io.getkey, 226  
 io.lines, 224  
 io.open, 223  
 io.popen, 226  
 io.read, 223, 225, 226  
 io.write, 224  
 io.writeline, 224  
 ipairs, 159  
 join, 114, 127  
 left, 131, 132  
 ln, 85  
 lngamma, 85  
 lower, 96  
 map, 127, 136  
 minus, 77, 114, 122, 128, 137  
 mod, 77, 87  
 mul, 77, 87  
 nand, 77  
 nor, 77  
 not, 105, 113, 114  
 notin, 77  
 ops, 187  
 or, 77, 104  
 pairs, 159  
 pop, 129  
 print, 54  
 protect, 184  
 purge, 116  
 put, 116  
 qsumup, 114  
 readlib, 48  
 registry.get, 213  
 replace, 95, 98  
 right, 131, 132  
 roll, 77  
 rtable.defaults, 203  
 rtable.forget, 204  
 rtable.get, 204  
 rtable.init, 204  
 rtable.mode, 204  
 rtable.purge, 204  
 rtable.put, 204  
 rtable.remember, 200  
 rtable.roinit, 204  
 seq, 123  
 setmetatable, 128, 132, 137, 205  
 settype, 124, 128, 131, 132, 191  
 sign, 85  
 sin, 85  
 sinh, 85  
 size, 96, 114, 122, 127, 136  
 sort, 114, 127, 136  
 split, 77, 95  
 sqrt, 85  
 squareadd, 77  
 strings.find, 97, 99  
 strings.match, 99  
 subset, 77, 104, 114, 122, 128, 137  
 sumup, 114  
 symmod, 77  
 tan, 85

tanh, 85  
 top, 127, 136  
 trim, 96  
 type, 127, 132, 136, 178  
 typeof, 124, 127, 132, 178  
 union, 77, 114, 122, 128, 137  
 unique, 114, 127, 136  
 unpack, 127, 136  
 upper, 96  
 with, 48  
 xnor, 77, 105  
 xor, 77, 104  
 xsubset, 77, 104, 114, 122  
 zip, 127, 137

---

## G

Garbage Collection, 56, 76, 214  
 Global Environment, 48

---

## H

Hardware  
     Keyboard, 225, 226

---

## I

I/O, 223  
     CSV Files, 227  
     dBASE Files, 227  
     INI Files, 227  
     Keyboard, 225  
     Locking Files, 226  
     Pipes, 226  
     SQLite Files, 227  
     Text Files, 223, 224  
     XML Files, 227  
 if Operator, 144, 145  
 if Statement, 61, 141  
     elif Clause, 141, 142  
     else Clause, 141, 142  
     on success Clause, 141, 143  
 import/alias Statement, 66  
 inc Statement, 83  
 INI Files  
     Reading & Writing Initialisation Files, 227  
 Initialisation, 48, 49, 198  
 Input  
     (please see I/O), 223

Input Conventions, 53  
 insert Statement, 59, 110, 126  
 Installation  
     DOS, 47  
     Linux, 44  
     Mac OS X, 48  
     OS/2 Warp 4 and later, 47  
     Solaris 10 & OpenSolaris, 43  
     UNIX Dependencies, 43, 44  
     Windows Binary Installer, 45  
     Windows Portable Edition, 46  
 Iterator, 158, 216

---

## K

Keywords, 72

---

## L

Libraries, 195, 197  
     import Statement, 66  
     Initialisation, 66  
     Initialisation Message, 198, 199  
     Initialisation Procedure, 199  
 library.agn, 48  
 Linked Lists, 228, 231  
     Doubly-linked, 230  
     Singly-linked, 229  
 Linux, 44  
 Logical Operators  
     and, 104  
     nand, 105  
     nor, 105  
     not, 105  
     or, 104  
     xnor, 105  
     xor, 104  
 Loops, 62, 149, 173, 192, 194  
     break Statement, 63, 154, 162  
     Conditional for Loops, 164  
     Control Variables, 156  
     Counting Backwards, 154  
     do/as Loops, 62, 151  
     do/od Loops, 151  
     do/until Loops, 151  
     for/as Loops, 63, 161  
     for/downto Loops, 155  
     for/in Loops, 155, 217  
     for/to Loops, 153  
     for/until Loops, 63, 161

- for/while Loops, 160
- Interruption, 180
- Iteration Over Procedures, 158, 217
- Iteration Over Sequences, 157
- Iteration Over Sets, 158
- Iteration Over Strings, 157
- Iteration Over Tables, 155
- Kahan-Babuška Summation, 154
- Kahan-Ozawa Summation, 154
- Key ~ Value Pairs, 156
- keys Keyword, 156
- Neumaier Summation, 154
- redo Statement, 163
- relaunch Statement, 163
- Round-Off Errors, 154
- skip Statement, 63, 162
- to/do Loops, 154
- while Loops, 149

Lua, 35

## M

- Mac, 48
- Maple
  - Maple V Release 3, 38
- Mapping & Zipping, 117
- Metamethods, 204
  - \_\_call, 212
  - \_\_iter, 212
  - Protecting, 210
  - Registry, 213
  - Weak References, 215
- Multisets, 71

## N

- Names, 72
- nargs, 176
- null, 56, 71, 76, 104, 155
- Numbers, 54, 71, 77, 82, 85, 179, 225
  - Abbreviations, 78
  - Billion, 78
  - Binary, 79
  - Dozen, 78
  - Hexadecimal, 79
  - Million, 78
  - Minus Zero, 79
  - Octal, 79
  - Percentage, 78
  - Scientific Notation, 78

- Thousand, 78
- Thousands Separator, 78

## O

- OOP
  - Methods, 219
- OpenSolaris, 43
- Operators
  - Logical, 105
  - Self-Defined, 219
  - Unary, 80, 85
- OS/2 Warp 4, 47
- Output
  - Printing Results, 53, 54, 193
  - Printing Tables, 106

## P

- Pairs, 61, 71, 77, 130, 179
  - Assignment, 61, 130
  - Colon Operator, 130
  - Indexing, 130
  - left & right Operators, 130
  - Operators & Functions, 132, 137
  - Read-Only, 209
- pop Statement, 126, 127, 128, 136
- Precedence, 77, 85
  - Associativity, 77
- Printing
  - print, 67
  - printf, 67
  - strings.format, 67
- Procedures, 64, 71, 158, 171, 179, 189
  - Arguments, 172, 176, 178
  - Closures, 216
  - Double Colon Notation, 179, 181
  - Error Handling, 178
  - Exception Handling, 184
  - Extending Built-in Functions, 214, 215
  - Global Variables, 175
  - Iterator Functions, 158, 216
  - Local Variables, 173, 192
  - Loops, 194
  - Metamethods, 204
  - Multiple Returns, 186
  - nargs, 176
  - Number of Arguments Passed, 176
  - Optional Arguments, 176, 178
  - Parameters, 171, 175

- Persistent Storage, 221
- Predefined Results, 203
- procname, 172
- Protected Mode, 184
- Remember Tables, 200
- Returning Procedures, 188
- Returns, 64, 171, 188
- Sandboxes, 194
- Scoping Rules, 192
- Shortcut Definition, 64, 189
- Summary, 223
- Type Checking, 178, 181, 183, 191
- varargs System Table, 176
- Variable Number of Arguments, 176
- Programmes, 65
  - Running, 65
  - Saving, 65

## R

- Registers, 133
  - create Statement, 135, 209
  - Creation, 133, 135
- Registry, 213, 231
- Regular Expressions
  - Lua-style, 98
- Remember Tables, 200
  - Functions, 204
  - Read-Only, 202
  - Standard, 200
- restart Statement, 55
- return Statement, 171
- rotate Statement, 129

## S

- Sandboxes, 194
- Scope, 192, 193, 216
  - Block, 192
  - scope Keyword, 193, 216
- Search
  - in Strings, 95, 96, 97, 98
  - in Structures, 113, 116, 117, 122, 127, 132, 136, 234
- Search and Replace
  - in Structures, 116, 117, 127, 136
- Sequences, 60, 71, 113, 116, 123, 130, 157, 179, 204, 225
  - Assignment, 60, 123

- bottom Operator, 129
- create Statement, 125, 130
- Creation, 125
- Deep Copying, 127, 132, 136
- delete Statement, 125
- Entries, 187
- Indexing, 123
- insert Statement, 125
- Insertion and Deletion, 125
- Operators & Functions, 128
- pop Statement, 126, 128
- Read-Only, 209
- Self-Reference, 126
- seq Operator, 123
- Size, 127, 136
- Sorting, 127, 136
- top Operator, 129
- Weak Ones, 215

- Sets, 60, 71, 113, 120, 130, 158, 179, 204, 225

- Assignment, 60, 120
- create Statement, 121
- Creation, 121
- Deep Copying, 122
- Operators, 122
- Read-Only, 209, 210
- Self-Reference, 121
- Size, 122

- Shell, 45, 47, 53, 65

- Short-Circuit Evaluation, 105

- Solaris, 43

- Sparc, 43

- Stack Programming, 128

- bottom Operator, 129
- duplicate Topmost Item, 130
- exchange Topmost Items, 130
- insert Statement, 128
- pop Operator, 129
- pop Statement, 129
- rotate Statement, 129
- top Operator, 129

- Statements

- Assignment, 73
- break Jump Control, 162
- case Condition, 147
- clear Deletion, 75
- create dict Initialisation, 113, 130
- create sequence Initialisation, 125, 130
- create set Initialisation, 130
- create table Initialisation, 111, 130
- dec Decrementation, 82
- delete Data Removal, 110, 126

- div Division, 83
- do/as Loop, 151
- do/od Loop, 151
- do/until Loop, 151
- duplicate Sequence Elements, 130
- enum Enumeration, 75
- exchange Sequence Elements, 130
- for/as Loop, 161
- for/in Loop, 155, 157, 158
- for/to Loop, 153
- for/until Loop, 161
- for/while Loop, 160
- if Condition, 141
- inc Incrementation, 82
- insert Data Entry, 110, 126
- insert Stack Item Entry, 129
- local Declaration, 173
- mul Multiplication, 83
- pop Stack Item Deletion, 128
- redo Jump Control, 163
- relaunch Jump Control, 162
- rotate Structure Elements, 129
- scope Statement, 193
- skip Jump Control, 162
- try/catch Error Interception, 185, 186
- unless Clause, 162, 173
- when Clause, 162, 172
- while Loop, 149
- stdin, stdout, stderr, 226
- Streams
  - stdin, stdout, stderr, 226
- Strings, 57, 71, 92, 157, 179, 206
  - ASCII Code, 96, 226
  - Checks, 97
  - Concatenation, 57, 77, 114
  - Empty Strings, 93
  - Escape Sequences, 94
  - ISO 8859/1 Latin-1, 103
  - Lower & Upper Case, 97
  - Multiline Strings, 92
  - Operators, 96
  - Pattern Matching, 98
  - Search & Replace Functions, 57, 95, 96, 97, 98, 103
  - Size, 96
  - Splitting into Words, 96
  - Substrings, 57
  - Trimming, 96
- Structures, 71
  - Weak Ones, 215
  - Write-Protection, 209
- Substrings, 57

- System Settings, 106
- System Variables, 48
  - \_G, 196
  - AGENAPATH, 45, 47, 48
  - ans, 55
  - environ.homedir, 49
  - io.stderr, 226
  - io.stdin, 226
  - io.stdout, 226
  - lasterror, 184
  - libname, 48, 49, 198
  - mainlibname, 48

## T

- Tables, 58, 71, 106, 113, 116, 118, 119, 130, 155, 179, 197, 200, 204, 225, 228, 231
  - Array Part, 113
  - Arrays, 107
  - Assignment, 58, 106, 107, 111
  - bottom Operator, 129
  - create Statement, 109, 130
  - Creation, 108, 109, 111
  - Cycles, 119
  - Deep Copying, 118
  - delete Statement, 110
  - Deletion, 110, 116, 117
  - Dictionaries, 111
  - Empty Tables, 109
  - Entries, 117, 187
  - Functions, 116
  - Hash Part, 113
  - Holes, 110, 115
  - Indexing, 59, 107, 108
  - insert Statement, 110
  - Insertion, 110, 116, 117
  - Key ~ Value Pairs, 112
  - Linked Lists, 228, 231
  - Nested Tables, 108
  - Operators, 115
  - pop Statement, 128
  - Read-Only, 209
  - References, 118, 228, 231
  - Removing Holes, 110
  - Self-Reference, 119
  - Size, 111
  - Sorting, 114
  - top Operator, 129
  - Unpacking Table Values by Name, 119, 167, 174

- Weak Ones, 215
- Tokens, 72
- try/catch Statement, 185, 186
- Types, 71, 133, 138, 181
  - Double Colon Notation, 181
  - Lightuserdata, 138
  - Threads, 138
  - Userdata, 138
  - User-Defined, 124, 131, 191

---

## U

- Unassignment, 56
  - clear Statement, 75
- UNIX, 48, 53, 65

---

## V

- Values
  - Defining new Variables within Procedures, 197
  - Reading Values within Procedures, 197

---

## W

- while Loops, 62, 149
- Windows, 45, 48, 53, 65
- with Statement, 167

---

## X

- XML
  - Dealing with SOAP Messages, 196
  - Reading XML Streams, 227
  - Writing XML Streams, 227