

agenda > >

# Crash Course

by Alexander Walz

# What is Agena ?

- Agena is an interpreted procedural programming language.
- It can be used in scientific, scripting, and many other applications.
- Its syntax looks like very simplified Algol 68 with elements taken from Maple, Lua and SQL, and some other languages.
- Binaries are available for Solaris, Mac OS X, Windows, OS/2 – eComStation, Linux, Raspberry Pi, and DOS.
- Agena is OpenSource, thus it is free.
- The implementation is based on the ANSI C sources of Lua 5.1.
- Sources and binaries are available at:

<http://agena.sourceforge.net>

# Contents, 1

- Installing Agena
- Running Agena
- First Steps
- Names & Assignment
- Data Types
  - Integral & Rational Numbers
  - Complex Numbers
  - Arithmetic
  - Strings

# Contents, 2

- Data Types, cont.
  - Boolean Expressions & Relations
  - Tables
  - Arrays
  - Dictionaries
  - Sets
  - Sequences & Registers
  - Pairs
- Control Statements
  - if Statements & is Operator
  - case Statements
  - onsuccess Clause

# Contents, 3

- Loops
  - for Loops
  - while Loops
  - do .. as, do .. until, and do .. od Loops
  - Combined for/while Loops
  - for/as and for/until Loops
  - Loop Control
- Procedures
  - Short-cut Procedures
  - Procedures
  - Local Variables
  - Variable Number of Arguments

# Contents, 4

- Procedures, cont.
  - Options
  - Type Checking
  - Error Traps
  - Predefined Results
  - Efficient Recursion
  - Functions as Binary Operators
  - Object-Oriented Programming
  - with and related Statements on Dictionaries

# Contents, 5

- Did you know ?
- Miscellaneous
  - Precedence
  - Mathematical Constants

agenda > >

Getting Started

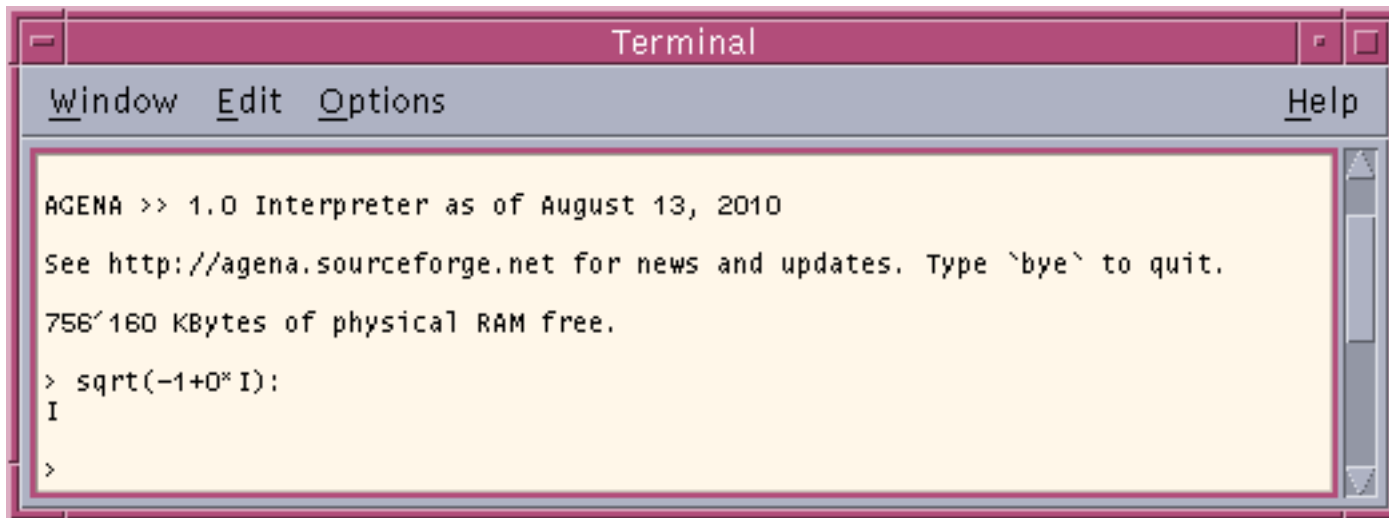


# Installing Agena

- In Solaris, OS/2 – eComStation, Linux, Windows, and Mac OS X, the respective installer automatically installs and sets up Agena. You do not have to add further settings yourself after installing the binaries.
- Information on how to install the DOS and Windows portable version is included in the manual or the respective read.me files.

# Running Agena

- In Windows and OS/2 - eComStation, simply click the **>>** icon in the programme group to start the interpreter.
- In Solaris, Linux, Mac and DOS, type `agena` in a shell.



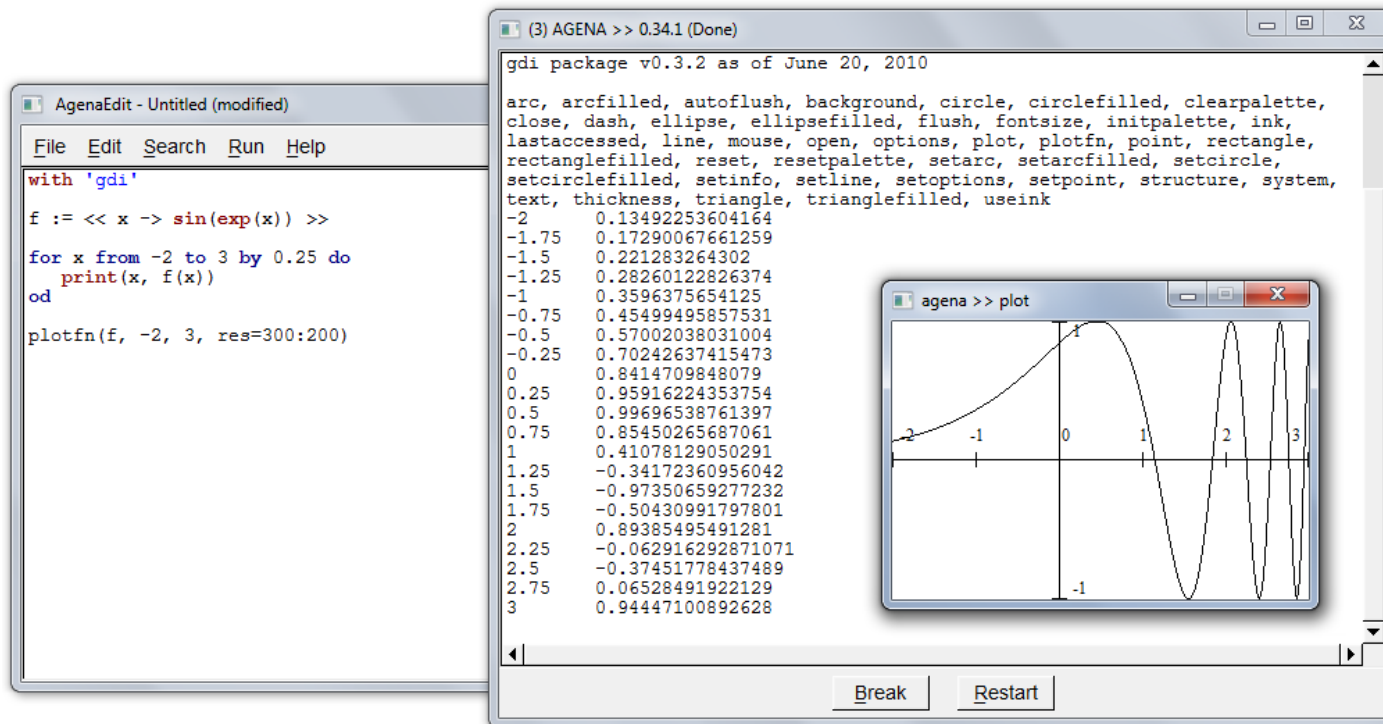
The screenshot shows a terminal window titled "Terminal" with a menu bar containing "Window", "Edit", "Options", and "Help". The terminal output is as follows:

```
AGENA >> 1.0 Interpreter as of August 13, 2010
See http://agena.sourceforge.net for news and updates. Type `bye` to quit.
756'160 KBytes of physical RAM free.
> sqrt(-1+0*I):
I
>
```

- Statements can be entered right after the '`>`' prompt.

# AgenaEdit, 1

- AgenaEdit is an editor providing syntax-highlighting and a runtime environment for Solaris, Mac, Linux, and Windows. It can be started by entering `agenaedit` in a shell.



# AgenaEdit, 2

- Type your programme in the editor window and press F5 to run it.
- Mark consecutive lines in your programme with a mouse or the keyboard and press F6 to execute only these lines.
- During computation, press the `break` button to interrupt the current computation.
- Press the `restart` button to clear all variables.
- Save or open your programmes using the `File` menu in the editor window.
- Just browse through the menu items for the other features.

# First Steps, 1

- Any valid Agena code can be entered at the console with or without a trailing colon or semicolon:
  - If an expression is finished with a colon, it is evaluated and its value is printed at the console. (This is not supported in AgenaEdit, use the print function instead.)
  - If the expression ends with a semicolon or neither with a colon nor a semicolon, it is evaluated, but nothing is printed.
- You may optionally insert one or more white spaces between operands in your statements.
- Assume you would like to add the numbers 1 and 2 and show the result. Just type:

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

# First Steps, 2

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

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

- Now the value 25 is stored to the name c, and you can refer to this number through the name c in subsequent calculations.
- Suppose that c is 25° Celsius. If you want to convert it to Fahrenheit, enter:

```
> 1.8*c + 32:  
77
```

- The cls statement clears the screen, restart clears all values, and bye quits the interpreter.

# Names & Assignment

- A name always begins with an upper-case or lower-case letter or an underscore, followed by one or more upper-case or lower-case letters, underscores or numbers in any order.
- Use the assignment operator := to store a value to a name.

```
> a := 1;  
  
> var1 := 'hello world';
```

- Delete a value by assigning it to null or use clear:

```
> a := null;  
  
> clear var1;
```

agenda > >

Data Types



# Integral & Rational Numbers

- Numbers can be represented like in the following examples.

- Integers:

```
> -1:  
-1
```

- More than one value can also be printed at one line:

```
> 0, 1, 1.0, 1, 1.0:  
0      1      1      1      1
```

- Rational numbers:

```
> 3.141592654, -1.0:  
3.141592654      -1
```

- Scientific notation:

```
> 10e-3, -1e3, 2.3e3:  
0.01      -1000      2300
```

# Complex Numbers

- There are two notations to represent complex numbers.

- The ! operator:

```
> 1!2, -1.1!-2, 3!0:  
1+2*I    -1.1-2*I    3
```

- The I operand:

```
> 1+2*I, -1.1-2*I, 3+0*I:  
1+2*I    -1.1-2*I    3
```

- Real part:

```
> real(1+2*I):  
1
```

- Imaginary part:

```
> imag(1+2*I):  
2
```

# Arithmetic, 1

- Agena allows to mix rational and complex numbers in calculations.
- Addition, subtraction, multiplication, division, and integer division:

rational	complex/mixed
$2 + 3$	$2+3*I + 1!2$
$2 - 3$	$2 - 3+1*I$
$2 * 3$	$2!2 * 3-I$
$2 / 3$	$2!0 / 3!1$
$2 \setminus 3$	$2!0 \setminus 3!1$

- Examples:

```
> 2+3, 2!0/3!1, 2 + 3!1:  
5          0.6-0.2*I          5+I
```

# Arithmetic, 2

- Modulus (for rational numbers only):

```
> 2 % 3:  
2
```

- Exponentiation with rational or integer power:

```
> 2 ^ 3.1, 2 ^ 3:  
8.5741877002903 8
```

- Exponentiation with integer power only (faster):

```
> 2 ** 3:  
8
```

# Strings, 1

- Strings can be enclosed in single or double quotes. There is no difference in meaning.

```
> 'this is a text':  
this is a text  
  
> "this is a text":  
this is a text
```

- Concatenation of two or more strings:

```
> 'Hello ' & 'world':  
Hello world
```

# Strings, 2

- Substrings:

```
> str := 'abcd';  
  
> str[2]:  
b  
  
> str[2 to 3]:  
bc  
  
> str[2 to -1]: # from 2nd two last character  
bcd  
  
> str[-1]: # last character  
d  
  
> str[-2 to -1]: # last two characters  
cd
```

# Boolean Expressions & Relations, 1

- Agena supports the logical values true and false, also called `booleans`. A third Boolean constant named fail indicates an error.
- Any condition, e.g.  $a < b$ , results to one of these logical values.
- Relational operators are:

Relation	Operator
less than	<
greater than	>
less or equal	<=
greater or equal	>=
equality	=
inequality	<>

# Boolean Expressions & Relations, 2

- Logical operators are:

Relation	Operator
Boolean and	and
Boolean or	or
Boolean complement	not
Boolean exclusive-or	xor

Relation	Operator
Boolean nand	nand
Boolean nor	nor

```
> 1 < 2:  
true  
  
> 1 < 2 and 1 = 0:  
false  
  
> true xor false:  
true
```



# Tables, 1

- 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.
- Tables can contain other tables, as well.

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

- To get the data with key 1, input:

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

# Tables, 2

- To get the second entry in the subtable, enter:

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

- There are two forms to create empty tables.

```
> tbl := [];  
  
> create table tbl;
```

- Tables can even be nested:

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

- The size operator returns the size of a table or any other structure.

# Arrays

- Tables with positive integral keys are called arrays.

```
> tbl := [10, 11, 12];
```

- Values can be inserted into arrays in two ways:

```
> tbl[4] := 'a'; tbl[5] := 'b';  
> insert 'a', 'b' into tbl;
```

- Values can be deleted like this:

```
> tbl[1] := null;  
> delete 'a', 'b' from tbl;
```

# Dictionaries

- Another form of a table is the *dictionary* which indices can be any kind of data - not only positive integers. Key-value pairs are entered with quoted keys and tildes, or with unquoted names and =.

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

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

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

- If a table key is a string, you can also use the notation:

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

# Sets, 1

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

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

- If you want to check whether 'donald' is part of the set s, just index it as follows:

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

# Sets, 2

- If you want to add or delete items to or from a set, use the insert and delete statements.

```
> insert 'daisy' into s;  
> delete 'daisy' from s;
```

- The in operator also checks whether an item is part of a set.

```
> 'donald' in s:  
true  
  
> 'daisy' in s:  
false
```

- Sets consume around 40 % less memory than tables.

# Sequences, 1

- Sequences can hold any number of items except null.

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

- You can access the items the usual way:

```
> s[2]:  
donald
```

- Values can be added as with tables.

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

# Sequences, 2

- Items can be deleted by setting their index position to null, or by applying delete.

```
> s[4] := null;  
> delete [1, 2, 2] from s;
```

- The in operator checks whether a sequence contains a given item.

```
> 'donald' in s:  
donald
```

- Sequences are twice as fast when adding values than tables.



# Registers, 1

- Registers are fixed-size arrays that also can store nulls.

```
> r := reg(null, 1, 'donald', true):  
reg(null, 1, donald, true)
```

- You can access the items the usual way:

```
> r[3]:  
donald
```

- If a value is deleted, the size of the register does not change:

```
> r[2] := null;  
  
> r:  
reg(null, null, donald, true)
```

# Registers, 2

- Registers have a pointer to the top of a register that can be changed so that data above the value of the top pointer can be hidden:

```
> registers.settop(r, 3); print(r, registers.gettop(r));  
reg(null, null, donald) 3
```

- Registers can be created with a predefined number of elements:

```
> create register r(8);  
  
> r:  
reg(null, null, null, null, null, null, null, null)
```

- The size of a register can be changed with the `registers.reduce` and `registers.extend` functions.

# Pairs

- Pairs hold exactly two values of any type (including null and other pairs).

```
> p := 10:11;
```

- The left and right operators provide read access to its left and right operands; the standard indexing method using integers is supported, as well:

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

- The left and right operand of a pair can be changed as follows:

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

agenda > >

Control Statements

# if Statement & if Operator

- 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 if operator checks a condition, too:

```
> result := if 1 < 2 then 'valid' else 'invalid' fi;

> result:
valid
```

# case Statements, 1

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

```
> c := 10;

> case c
>   of -1 then          # one value to be compared
>     print('negative')
>   of 0, 1 then        # multiple values to be compared
>     print('non-negative')
>   of 2 to infinity   # a range
>     print('non-negative, too')
>   else
>     print('negative, too')
> esac;
non-negative, too
```

# case Statements, 2

- A variant works like the if statement and may improve readability of code.

```
> x := 10;

> case
>   of x < 0 then return -1
>   of x = 0 then return 0
>   else return 1
> esac
1
```

# onsuccess Clause

- Both if and case statements support an optional onsuccess clause. If at least one of the conditions evaluated to true, then the statements in the onsuccess clause are also executed.

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

> case c
>   of 'agena' then
>     print('Agena !')
>   of 'lua' then
>     print('Lua !')
>   onsuccess
>     flag := true
>   else
>     print('Another programming language !')
> esac;
Agena !

> flag:
true
```



agenda > >

Loops

# for Loops, 1

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

# for Loops, 2

- 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.

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

# for Loops, 3

- The value of the control variable can be accessed outside the loop.
- Since after the last iteration, the control variable is internally increased by the step size a very last time, its contents is:

```
> for i to 3 do  
>   result := i^2  
> od;  
  
> i:  
4
```

# for Loops, 4

- A for/in loop iterates over all values in a table, set, and sequence. With strings, it iterates over each character from the left to the right.

```
> for i in ['Agena', 'programming', 'language'] do
>   print(i)
> od
Agena
programming
language

> for i in 'Agena' do print(i) od
A
g
e
n
a
```

# for Loops, 5

- You can also iterate only over the keys of a table (or sequence) or both keys and values:

```
> for keys i in ['donald' ~ 'duck', 'daisy' ~ 'duck'] do
>   print(i)
> od;
daisy
donald

> for i, j in ['donald' ~ 'duck', 'daisy' ~ 'duck'] do
>   print(i, j)
> od;
daisy  duck
donald duck
```

# 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.
- 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
```

# do .. as & do .. until Loops

- Variations of while are the do .. as and do .. until loops which check a condition at the end of the iteration.
- Thus – contrary to while loops - the loop body will always be executed at least once.

```
> c := 0;          c := 0
> do              > do
>   inc c         >   inc c
> as c < 10;     > until i = 10;
> c:             > c:
10              10
```



# do .. od Loops

- Infinite loops are supported by do .. od loops, a syntactic sugar for `while true do .. od`.

```
> c := 0;

> do
>   inc c;
>   if c > 9 then break fi
> od;

> c:
10
```

- See the `Loop Control` sheet on how to exit these loops.

# Combined for/while Loops

- 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
```

# for/until and for/as Loops

- for loops can also be combined with a closing until or as condition.

```
> for x to 10 do  
>   print(x)  
> as i < 3;  
1  
2  
3
```

```
> for x to 10 do  
>   print(x)  
> until i = 3;  
1  
2  
3
```

# Loop Control, 1

- Agena features three statements to control loop execution. The following two are applicable to all loop types.
  - The skip statement causes another iteration of the loop to begin at once, thus skipping all of the following loop statements after the skip keyword for the current iteration.
  - The break statement quits the execution of the loop entirely and proceeds with the next statement right after the end of the loop.

```
> for i to 5 do
>   if i = 3 then skip fi;
>   print(i);
>   if i = 4 then break fi
> od;
1
2
4
```

# Loop Control, 2

- skip and break can also be combined with the when condition:

```
> for i to 5 do
>   skip when i = 3;
>   print(i);
>   break when i = 4
> od;
1
2
4
```

# Loop Control, 3

- The redo statement restarts the current iteration of a for/to or for/in loop from its beginning, without incrementing the loop control variable or processing the next item in a structure.

```
> flag := true;

> for i to 3 do
>   print(i);
>   if flag and i = 2 then
>     flag := false;
>     redo
>   fi
> od;
1
2
2
3
```

# Loop Control, 4

- The relaunch statement, however, restarts a for/to or for/in loop completely.

```
> flag := true;

> for i to 3 do
>   print(i);
>   if flag and i = 2 then
>     flag := false;
>     relaunch
>   fi
> od;
1
2
1
2
3
```

agenda > >

Procedures



# Short-cut Procedures

- 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.
- Let us define a simple factorial function with one argument.

```
> factorial := << (x) -> exp(lngamma(x+1)) >>;  
  
> factorial(4):  
24
```

- A function with two arguments:

```
> sum := << (x, y) -> x + y >>;  
  
> sum(1, 2):  
3
```

# Procedures

- Let us write a procedure to compute the factorial of an integer.
- A procedure can call itself to generate the final result.
- The return statement passes the result of a computation.

```
> factorial := proc(n) is
>   # computes the factorial of an integer n
>   if n < 0 then return fail
>   elif n = 0 then return 1
>   else return factorial(n-1)*n
>   fi
> end;

> factorial(4):
24
```

# Local Variables

- A local variable is known only to the respective procedure and the block where it has been declared.
- It cannot be used in other procedures, the interactive Agena level, or outside the block where it has been declared.

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

> factorial(10):
3628800
```

# Variable Number of Arguments

- If you want to pass a variable number of arguments, use the ? keyword in the parameter list.
- The varargs system table contains all variable arguments passed with the ? facility. Values can be accessed like with any other table.
- The system variable nargs contains the number of arguments passed (both with the ? facility and without).

```
> f := proc(?) is
>   return nargs, varargs, varargs[1]
> end;

> f('Beowulf', 'Grendel'):
2      [Beowulf, Grendel]      Beowulf
```

# Options, 1

- A function does not have to be called with exactly the number of parameters given at procedure definition.
- You may optionally pass less or more values at run-time. If no value is passed for a parameter, then this parameter is automatically set to null at function call.

```
> f := proc(a, b, c) is
>   return a, b, c
> end;

> f(1):
1      null      null
```

- If you pass more arguments than there are actual parameters, excess arguments are ignored.

# Options, 2

- 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
```

# Options, 3

- If the left-hand value of the pair in a function call shall denote a string, you can spare the single quotes put between the string by using the = token which converts the left-hand name to a string.

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

# Type Checking, 1

- You can check the type of arguments passed in two ways:
- Query the type with the `::` or `:-` (the negation) operators:

```
> f := proc(x) is
>   if x :- number then error('no number argument') fi;
>   return x
> end;

> f('men ne cunnon hwyder helrunan hwyrftum scriþað.'):
wrong type of argument
```

- State the expected type in the parameter list:

```
> f := proc(x :: number) is
>   return x
> end;

> f('men ne cunnon hwyder helrunan hwyrftum scriþað.'):
Error in stdin:
  invalid type for argument #1: expected number, got string.
```



# Type Checking, 2

- Up to four types may be given:

```
> f := proc(x :: {number, complex}) is
>   return tostring(x)
> end

> f(1!2)
1      2
```

- Besides checking the arguments, the return can also be insured:

```
> f := proc(x :: number) :: number is
>   return tostring(x)
> end

> f(1)
Error in stdin, at line 2:
  `return` value must be of type number, got string.
```

# Error Traps

- The try/catch statement catches errors:

```
> success, s := true, null;

> try
>   print(s[1]) # provoke an error by indexing null
> catch msg then
>   success := false
> yrt;

> success:
false
```

- Alternatively, the protect function also traps errors.

# Predefined Results

- Predefined results can be set with the `rtable.defaults` function by entering them into a remember table.
- Agena returns the given predefined result if it exists and does not compute it by executing the procedure body, so there is also an increase in speed.

```
> rtable.defaults(fact, [ # defaults for fact(0) .. fact(3)
>   -1~undefined, 0~1, 1~1, 2~2, 3~6
> ]);

> fact(-1):
undefined

> rtable.defaults(fact):
[[2] ~ [2], [1] ~ [1], [0] ~ [1], [3] ~ [6], [-1] ~ [undefined]]
```

# Efficient Recursion

- Agena remembers procedure results if the `rtable.remember` function is invoked. An optional table of predefined results can also be given. This speeds up recursive procedures significantly.

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

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

> fib(50):
20365011074
```

- For the differences between defaults and `remember` check the manual (Chapter 7.23). Chapter 6.18.1 describes the feature `reminisce` shortcut.

# Functions as Binary Operators

- An ordinary function of two arguments can be called just like a binary operator.

```
> plus := proc(x, y) is return x + y end;
```

```
> 1 plus 2:
```

```
3
```

- When using a function this way, it has always the highest precedence.

# Object-Oriented Programming, 1

- Methods for tables can be implemented OOP-style using the @@ syntax:

```
> account := ['balance' ~ 0];  
  
> proc account@@deposit(x) is  
>   inc self.balance, x;  
> end;  
  
> account@@deposit(100)  
  
> account.balance:  
100  
  
> proc account@@withdraw(x) is  
>   dec self.balance, x  
> end;
```

# Object-Oriented Programming, 2

- A constructor that created new accounts:

```
> proc account@@new(o) is
>   setmetatable(o, self);
>   self.__index := self;
>   return o
> end;

> a := account@@new(['balance' ~ 0]);

> a.balance:
0
```

# Object-Oriented Programming, 3

- Inheritance: here we define a new account class based on the one defined above that does not allow overdrafts.

```
> creditaccount := account@@new();

> proc creditaccount@@withdraw(x) is
>   if x > self.balance then error('Error, not enough credit.') fi;
>   dec self.balance, x;
>   return self.balance
> end;

> b := creditaccount@@new();

> b@@withdraw(1000):
Error, not enough credit.
```

- For more information, please check Chapter 6.24 of the Primer and Reference.



# with and related Statements on Dictionaries, 1

- The with statement unpacks values from a dictionary, declares them local and can access them in a block. The new names are variables on their own and do not refer to the indexed values in the table. The in assignment spares some typing.

```
> zips := [duedo = 40210, bonn = 53111, cologne = 50667];

> with duedo, bonn in zips do
>   print(duedo, bonn, cologne);
>   duedo := null # zips.duedo is not changed
> od;
40210      53111      null

> zips.duedo:
40210

> duedo, bonn in zips; # equals duedo, bonn := zips.duedo, zips.bonn

> duedo, bonn:
40210      53111
```

# with and related Statements on Dictionaries, 2

- A flavour of the with statement allows to reference an entry by just an underscore. It also allows to actively change values in the table.

```
> zips := [duedo = 4000, bonn = 5300]

> with zips do
>   print(_.bonn);
>   _.bonn := 53111
> od
5300

> zips:
[bonn ~ 53111, duedo ~ 4000]
```

agenda > >

Did you know ?

# Did you know, 1 ?

- You can send and receive data on the TCP level across the Internet and LANs with the net package.
- You can load your own programmes into an Agena session by using the run function (e.g. `run 'progrname.agn'`) or starting Agena from the shell with `agena -i progrname.agn`.
- The map function applies a function to all elements of a table, set, or sequence, e.g. `map(<< x -> x^2 >>, [2, 3]) → [4, 9]`. You may also try `countitems`, `remove`, `select`, `subs`, and `zip`.
- If you want your self-written procedures, constants, etc. to be available every time you invoke the interpreter, just put them into a file called `agena.ini` file (Windows, OS/2, DOS) or `.agenainit` (UNIX, Mac, Haiku) in your home directory.

## Did you know, 2 ?

- Data you compute in a session can be stored to a file using the `save` function to be read into another session later by `read`.
- The way Agena outputs tables, sets, sequences, complex numbers, and pairs can be changed by modifying the `environ.aux.print*` procedures in the `library.agn` file located in the `lib` directory of your Agena installation.
- Data stored in CSV and XML files can be imported with the `xml` package or the `utils.readcsv` and `utils.readxml` functions.
- Errors issued by Agena, preventing programmes to finish successfully, can be intercepted with `protect`.
- If you do not like the default prompt, just enter something like:

```
_PROMPT := '% '
```

agenda > >

Miscellaneous

# Precedence

- Operator precedence follows the table below, from lowest to highest.

Prio	Operators
10	or xor nor
9	and nand
8	< > <= >= = == <> ~= ~<> :: :-
7	in subset xsubset union minus intersect atendof
6	& : @ \$
5	+ - split    ^^
4	* / % \ <<< >>> <<<< >>>> && *% /% +% -%
3	not -
2	^ **
1	! ~~ and all other unary operators

# Packages, 1

- Agena features various packages that can be invoked with the import statement, e.g. `import calc alias`.

Package	Function
ads	Database specialised on storing and retrieving strings
bags	Multisets, Cantor sets that count occurrences
astro	Astronomical time and date functions
binio	Functions for processing binary files
calc	Undergraduate Calculus package
clock	Functions to process hours, minutes, and seconds
cordic	CORDIC numeric functions
div	Fractions
environ	Access to the Agena environment



# Packages, 2

Package	Function
fractals	Various fractals & plotting routines, some FRACTINT support
gdi	Graphics
gzip	Read and Write UNIX gzip compressed files
hashes	String hashes
io	Input/output functions for console and files
linalg	Undergraduate Linear Algebra
llist	Linked lists
mapm	Mathematical arbitrary precision library for the real domain
math	Additional mathematical functions
net	IPv4-based exchange of data over the Internet or LANs
numarray	Numeric C arrays

# Packages, 3

<b>Package</b>	<b>Function</b>
os	Functions to operate with the underlying operating system
registers	Functions for register administration
registry	Functions to access the registry
rtable	Administration of remember tables
skycrane	Various easy-to-use wrappers to Agena functions
stats	Statistical functions
strings	Various string handling functions
tables	Functions specialised on table processing
tar	Functions to list, read, and extract UNIX tar archives
utils	Utility functions, e.g. CSV import and export
usb	libusb binding

# Packages, 4

Package	Function
xbase	xBase file support (i.e. dBASE <sup>(tm)</sup> III+)
xml	XML decoding (LuaExpat)

# Mathematical Constants

- Agena features the following numeric constants:

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

# Any Questions ?

- For further information, please consult
  - the *Primer and Reference*, a manual explaining Agena on 604 pages
  - the *Quick Reference*, an overview of all the functions available

- Both are available at

<http://agena.sourceforge.net/documentation.html>

- Forum:

<http://sourceforge.net/projects/agena/forum>

The image shows a spreadsheet titled 'Basic Operators and Functions'. The table has four columns: Name, Operator, Function, and Functionality. The rows list various operators and their corresponding functions and what they do.

Name	Operator	Function	Functionality
abs	x		on a number, abs returns its absolute value; on a string, it returns its length; on a boolean, returns 0 for false, and 1 for true, w
allnames		x	returns all names assigned in a session
assume		x	issues an error, if its condition is false
atop		x	returns various information on the size of structures
bye			quits an interactive session
clear	x		unassigns a name and garbage-collects its former value
concat		x	concatenates strings with an optionally given delimiter
error		x	terminates execution of a function and issues an error
filled	x		checks whether a structure contains at least one non-null
gc		x	initiates or administers garbage collection
getenv		x	Returns the current environment in use by a function
globals		x	determines whether function includes global variables
getmetatable		x	returns the metatable of a structure
gettype		x	returns the user-defined type of a structure or procedure
has		x	checks whether a structure contains an element
hasstable		x	checks whether a function has a remember table
isnull	x		checks whether a function has a remember table
left	x		returns the left operand of a par
load		x	loads a chunk using a function to get its pieces