agena >>

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 – ArcaOS, Linux, Raspberry Pi, and DOS.

- Agena is Open Source, 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

agena >>

Contents, 1

- Installing Agena

- Running Agena

- First Steps

- Names & Assignment

- Data Types

  - Integral & Rational Numbers

  - Complex Numbers

  - Arithmetic

  - Strings

agena >>

Contents, 2

- Data Types, cont.
  - Boolean Expressions & Relations
  - Tables
  - Arrays
  - Dictionaries
  - Sets
  - Sequences & Registers
  - Pairs

- Control Statements
  - if Statements & if Operator
  - case Statements
  - onsuccess Clause

agena >>

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
    - Procedures
    - Local Variables
    - Variable Number of Arguments
    - Options

agena >>

Contents, 4

- Procedures, cont.
    - Error Handling & Error Traps
    - Type Checking
    - Predefined Results
    - Efficient Recursion
    - State Tables
    - Functions as Binary Operators
    - Short-cut Procedures
    - Object-Oriented Programming
    - with and Related Statements
    - Syntactic Sugar

agena >>

Contents, 5

- Did you know ?

- Miscellaneous
    - Precedence
    - Mathematical Constants

agena >>

Getting Started

Installing Agena

- In Solaris, OS/2 – ArcaOS, 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 - ArcaOS, simply click the **>>** icon in the programme group to start the interpreter.

- In Solaris, Linux, Mac and DOS, type `agena` in a shell.
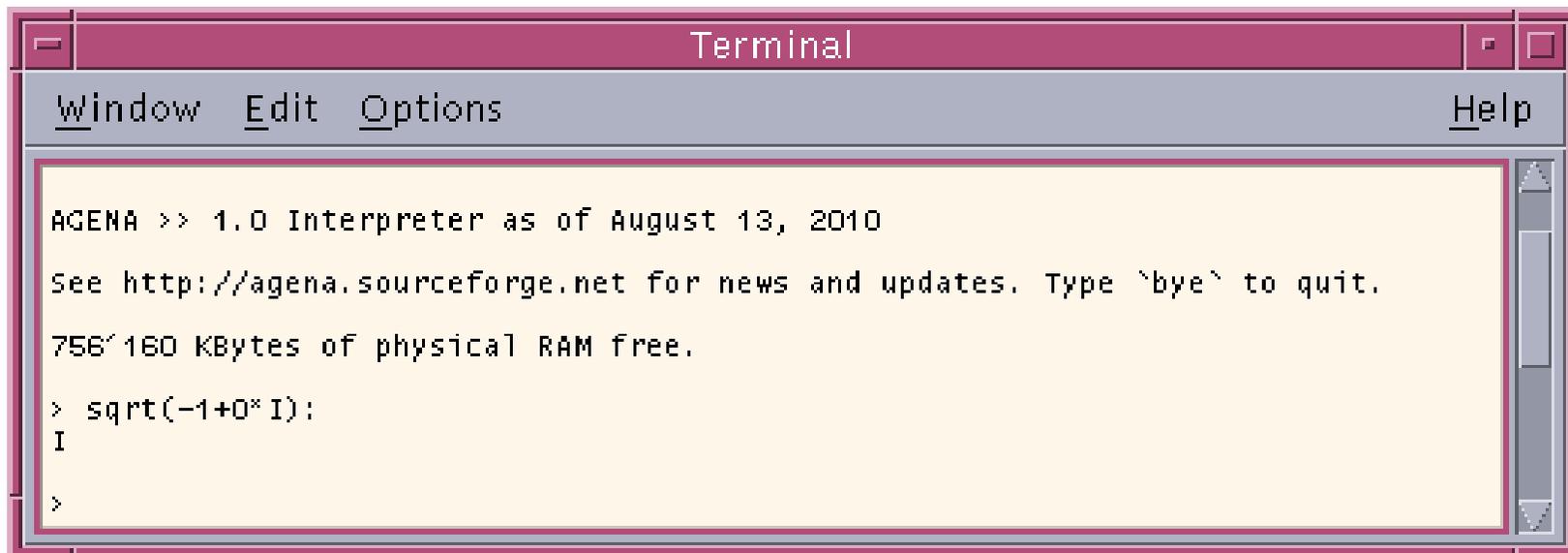
```
Terminal

Window   Edit   Options                                    Help

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.

agena **>>**

- 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.
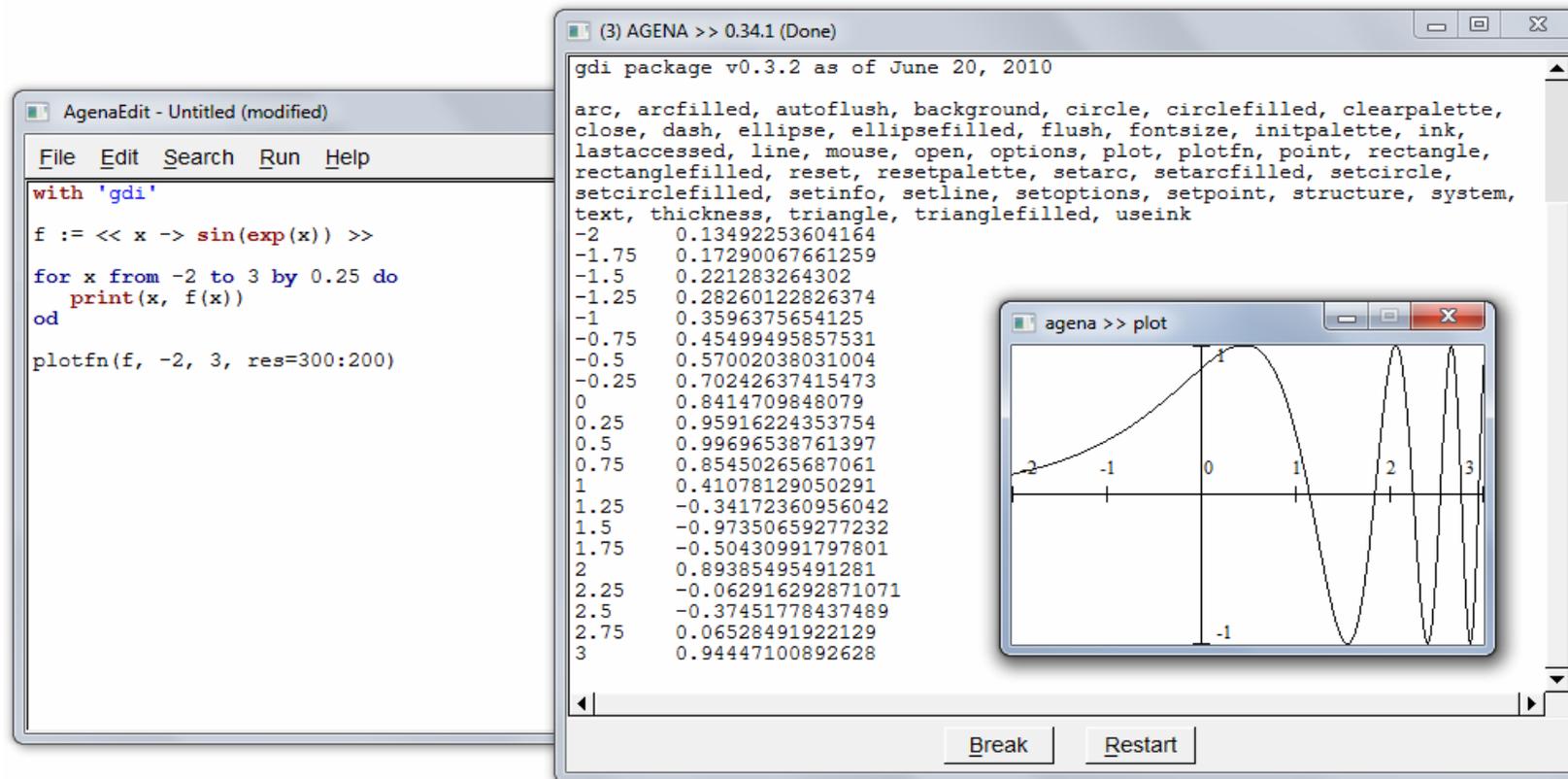
AgenaEdit, 2

- 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.

agena >>

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 will be evaluated and its value 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 will be 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
```

agena >>

- 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.

agena >>

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;
```

agena >>

- Compound assignment is supported in three fashions:

```
> a := 1;

> inc a;

> a:
2
```

```
> a := 1;

> inc a, 2;

> a:
3
```

```
> a := 1;

> a +:= 1;

> a:
2
```

```
> a := 1;

> a +:= 2;

> a:
3
```

```
> c := 1;
> a := c++;

> a, c:
1      2

> a := c--;

> a, c:
2      1
```

Function	Statement	Compound
Addition	inc	+:=
Subtraction	dec	-:=
Multiplication	mul	*:=
Division	div	/:=
Modulo	mod	%:=

```
> c := 1;
> a := ++c;

> a, c:
2      2

> a := --c;

> a, c:
1      1
```

agena >>

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
```

agena >>

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
```

agena >>

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 \ 3	2!0 \ 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
```

agena >>

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

> a := 'Hello ';

> a &:= 'World';

a:
Hello World
```

agena >>

Strings, 2

- Substrings:

```
> str := 'abcd';

> str[2]:
b

> str[2 to 3]:
bc

> str[2 to -1]:    # from 2nd to last character
bcd

> str[-1]:         # last character
d

> str[-2 to -1]:   # last two characters
cd
```

agena >>

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	<>

agena >>

Boolean Expressions & Relations, 2

- Logical operators are:

Relation	Operator
Boolean and	and
Boolean or	or
Boolean complement	not

Relation	Operator
Boolean nand	nand
Boolean nor	nor
Boolean exclusive-or	xor

```
> 1 < 2:
true

> 1 < 2 and 1 = 0:
false

> true xor false:
true
```

agena >>

Boolean Expressions & Relations, 3

- If you add, subtract, multiply or divide a Boolean - or a relation that evaluates to a Boolean - with a number, then `true` will represent number 1 and `false` 0.

- Thus, you can comfortably write statements without having to use `if` conditions, for example:

```
> return (x > 0)*x;

> return if x > 0 then x else 0 fi;
```

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],        long form
>    3 ~ ['c', 7.59]
> ];
```

```
> tbl := [
>    ['a', 7.71],
>    ['b', 7.70],        short form
>    ['c', 7.59]
> ];
```

- To get the data with key 1, input:

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

agena >>

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.

agena >>

Tables, 3

- To select a sequence of elements in a table, use the to notation:

```
> tbl[1 to 2]:
[[a, 7.71], [b, 7.7]]
```

- When trying to index a null value with square brackets, Agena returns an error. When using curly brackets, however, Agena just returns null.

```
> tbl := null;

> tbl[1]:
Error in stdin at line 1:
   attempt to index global `tbl` (a null value) with a number value

Stack traceback:
   stdin, at line 1 in main chunk

> tbl{1}, tbl{1 to 2}:
null    null
```

agena >>

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;
```

agena >>

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
```

agena >>

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.

agena >>

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;
```

agena >>

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 will not change:

```
> r[2] := null;

> r:
reg(null, null, donald, true)
```

agena >>

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;
```

Control Statements

if Statement, 1

- 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
```

- A short form is also available if only one statement shall be executed and no else clause is needed:

```
> 1 < 2 ? print('valid')
valid
```

agena >>

- If statements also support simple assignments in the conditions, even in elif clauses. If the right-hand side evaluates to neither false, fail nor null, then the corresponding then part will be executed.

- Compare:

```
> flag := io.read();
> if flag then
>    print(flag)
> fi;

> if flag := io.read() then
>    print(flag)
> fi;
```

agena >>

if Operator, 1

- The if operator checks a condition and returns the result:

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

> result:
valid
```

- 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;

> b := with m, n := x, 2*x -> if x < 0 then m else n fi;
```

- You can also add one or more elif clauses.

if Operator, 2

- The extended version of the if operator is similar to the if statement. Note the sequence `if is` and the obligatory return expressions in the bodies; elif's and else's are optional, as are the statements in the bodies.

```
> 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');  # just one statement
>           return 0
>        else                         # no statement
>           return 1
>        fi;

> sgn:
1
```

agena >>

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')
>    esle               # this keyword is optional, just a beautifier
> esac;
non-negative, too
```

agena >>

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
```

agena >>

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
```

agena >>

Alternative end Clause

- You can now use the **end** token instead of the closing **fi**, **od**, **esac**, **yrt** and **epocs** keywords. Examples:

```
> if 1=1 then print(true) else print(false) end;

> for i to 10 do
>    print('Agena !')
> end;
```

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
```

agena >>

for Loops, 2

- The from and step clauses are optional.

- If the from clause is omitted, the loop will start with the initial value 1.

- If the step clause is omitted, the step size will be 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
```

agena >>

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 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, 1

- A while loop first checks a condition and if this condition is true or any other value except false, fail, or null, it will iterate 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
```

agena >>

while Loops, 2

- A simple assignment can also be done in the while condition. This allows for shorter code. If the right-hand side evaluates to neither false, fail or null, then the loop body will be executed.

- Just compare the following two statements.

```
> flag := true;
> while flag do
>    flag := io.read();
>    if flag = 'Z' then break fi
> od

> while flag := io.read() do
>    if flag = 'Z' then break fi
> od
```

agena >>

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 Support 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.

agena >>

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
```

- Likewise, the until condition quits the loop:

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

agena >>

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 x < 3;
1
2
3

> for x to 10 do
>     print(x)
> until x = 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
```

Procedures

Procedures, 1

- 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.

- The procname keyword is substituted by the name with which the procedure was invoked.

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

> factorial(4):
24
```

agena >>

Procedures, 2

- Alternatively, a function can be defined with the procedure statement.

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

> factorial(4):
24
```

You can put the local keyword before the procedure keyword to define local procedures.

agena >>

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
```

agena >>

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 will automatically be 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 will be ignored.

agena >>

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 will be chosen, 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
```

agena >>

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
```

Error Handling & Error Traps

- The error function issues an error:

```
> if 1 = 1 then error('Oops !') fi
Oops !

Stack traceback: in `error`
   stdin, at line 1 in `(null)` in `(null)`
```

- 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 traps errors, as well.

agena >>

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:

```
> procedure f (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.
```

agena >>

Type Checking, 2

- Up to five types may be given when putting them in curly brackets:

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

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

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

```
> 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.
```

agena >>

- Numbers can be examined further with the keywords

  - `integer` (any integral number),

  - `posint` (positive integer),

  - `nonnegint` (nonnegative integer),

  - `positive` (positive floats and integers),

  - `nonnegative` (nonnegative floats and integers).

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

> f(Pi)
Error in stdin:
   type integer expected for argument #1, got number.
```

agena >>

Type Checking, 4

- Function arguments can be checked further with the pre clause …

```
> f := proc(x :: number) is
>    pre x > 0 is
>      return x
> end

> f(0):
In stdin at line 2:
   Error in pre-condition: posture not satisfied.
```

- … and the result with the post clause:

```
> f := proc(x :: number) is
>    return post x > 0 with x
> end

> f(0)
In stdin at line 2:
   Error in post-condition: invalid return.
```

agena >>

Predefined Results

- Predefined results can be set with the rtable.defaults function by entering them into a remember table.

- Agena will return 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 will remember 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 14.4). Chapter 6.18.1 describes the feature reminisce shortcut.

agena >>

State Tables

- A table can be assigned to a function with the **store** feature. This internal table will remain active during a whole session and you can read or write values to it in subsequent calls to the function.

```
> f := proc() is
>    feature store
>    store[1] := Pi
>    store.entry := E
>    return store, store[1], store.entry
> end;

> f():
[1 ~ 3.1415926535898, entry ~ 2.718281828459]   3.1415926535898
2.718281828459
```

agena >>

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.

agena >>

Short-cut Procedures, 1

- 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
```

agena >>

Short-cut Procedures, 2

- The `with` clause allows to define local variables.

```
> fact := << (x :: number)
>    with n := 1
>    -> exp(lngamma(x + n)) >>;

> fact := << (x :: number)
>    with m, n := 0, 1
>    -> exp(lngamma(x + n)) + m >>;
```

- Alternatively, you can define a function with the def or define statement:

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

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

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;
```

agena >>

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.

agena >>

- The with statement 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 right between the with and in tokens are automatically written back to the table.

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

> with duedo, cologne in zips do
>    duedo   := 40237;  # change duedo entry
>    cologne := null     # cologne will be deleted ☺
> od;

> zips:
[duedo = 40237, bonn = 53111]

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

> duedo, bonn:
40237   53111
```

agena >>

with and Related Statements, 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]
```

agena >>

with and Related Statements, 3

- Yet another variant allows to easily define local variables to be used in a block:

```
> with a, b := 1, 2 do
>     c := a + b
> od;

> print(a, b, c):
null    null    3
```

Syntactic Sugar

- Just an overview of some syntactic sugar available:

```
> break when x <> 0;
> if x <> 0 then break fi;

> skip when x <> 0;
> if x <> 0 then skip fi;

> return when x <> 0;
> if x <> 0 then return fi;

> return when x <> 0 with y;
> if x <> 0 then return y fi;
```

Did you know ?

Did you know, 1 ?

- If you do not like the default prompt, just enter something like:

  `_PROMPT := '% '`

- You can load your own programmes into an Agena session by using the run function (e.g. `run 'progname.agn')` or starting Agena from the shell with `agena -i progname.agn`.

- If you want your self-written procedures, constants, etc. to be available every time you invoke the interpreter, just put them into an agena.ini file residing in your home directory.

- Data you compute in a session can be stored to a file using the save function, to be read into subsequent session later by read.

- You can send and receive data on the TCP level across the Internet and LANs with the net package.

agena >>

Did you know, 2 ?

- Data stored in CSV and XML files can be imported with the utils.readcsv and utils.readxml functions. See xml package, too.

- 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.

agena >>

Miscellaneous

Precedence

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

Prec	Operators
10	`or xor nor xnor`
9	`and nand`
8	`< > <= >= = == <> ~= ~<> :: :- |`
7	`in notin subset xsubset union minus intersect atendof |-`
6	`& : @ $ $$`
5	`+ - || ^^ split inc dec`
4	`* / % \ <<< >>> <<<< >>>> && *% /% +% -% %% symmod mul div intdiv mod`
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
aconv	GNU iconv port, to transform strings between codepages
ads	Database specialised on storing and retrieving strings
bags	Multisets, Cantor sets that count occurrences
bimaps	Bi-directional maps
astro	Astronomical time and date functions
binio	Functions for processing binary files
bytes	Bits and bytes twiddling
calc	Undergraduate Calculus package
clock	Functions to process hours, minutes, and seconds

agena >>

Packages, 2

Package	Function
com	RS-232 communication via COM ports
cordic	CORDIC numeric functions
div	Fractions
dual	Dual numbers
environ	Access to the Agena environment
fastmath	Numeric approximations
fractals	Various fractals & plotting routines, some FRACTINT support
gdi	Graphics
gzip	Read and Write UNIX gzip compressed files
heaps	Skewed & binary heaps plus AVL trees
hashes	String hashes

Packages, 3

Package	Function
io	Input/output functions for console and files
linalg	Undergraduate Linear Algebra
llist	Linked lists
long	80-Bit Floating-Point Arithmetic
mapm	Mathematical arbitrary precision library for the real domain
math	Additional mathematical functions
memfile	String memory files
mp	GNU Multiple Precision Arithmetic Library (GMP)
mpf	GNU Multiple Precision Floating-Point Reliable Library (MPFR)
net	IPv4-based exchange of data over the Internet or LANs
numarray	Numeric C arrays

agena >>

Packages, 4

Package	Function
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
sema	Unique integer IDs
skycrane	Utilities and easy-to-use wrappers to some 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

agena >>

Packages, 5

Package	Function
xbase	xBase file support (i.e. dBASE [tm] III+)
xml	XML decoding (LuaExpat port)
zx	Sinclair ZX Spectrum numeric functions

agena >>

Mathematical Constants

- Agena features the following numeric constants, some of them are:

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

- See also Chapter A3 of the Primer and Reference.

agena >>

Any Questions ?

- For further information, please consult

  - the *Primer and Reference*,
    a manual explaining Agena on 970 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