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, Linux, 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

Contents, 2

- Data Types, cont.
  - Boolean Expressions & Relations
  - Tables
  - Arrays
  - Dictionaries
  - Sets
  - Sequences
  - 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
- Did you know ?
- Miscellaneous
  - Precedence
  - Mathematical Constants

Getting Started

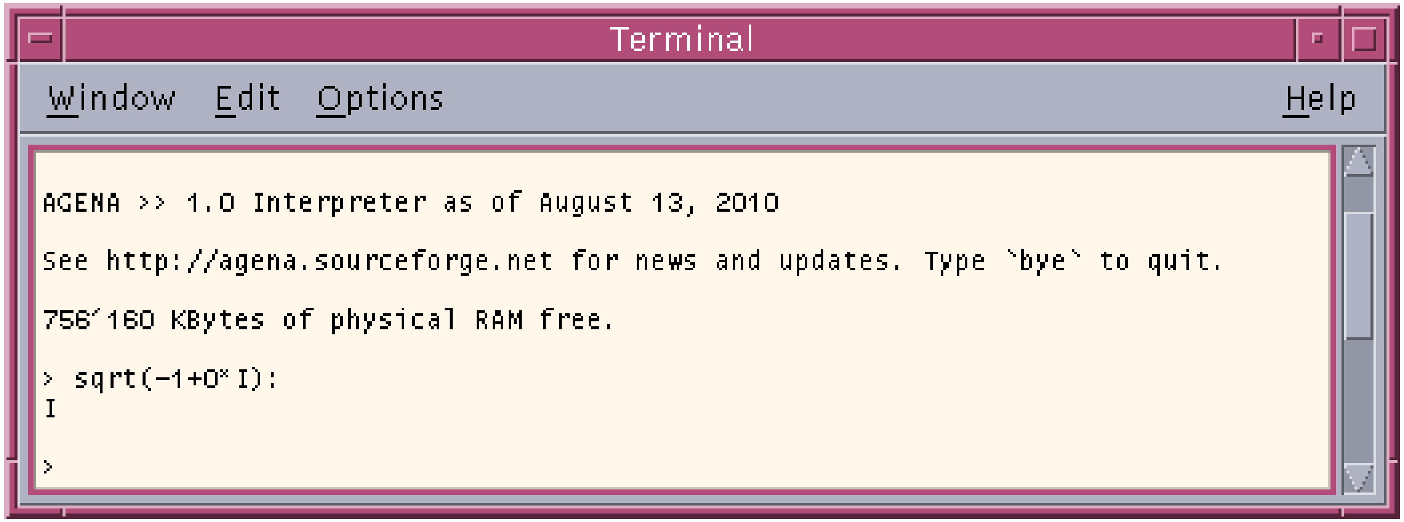
Installing Agena

- In Solaris, 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 version is included in the manual.
- Agena has been compiled successfully in OS/2 and Haiku in the past.

Running Agena

- In Windows, simply click the >> icon in the programme group to start the interpreter.

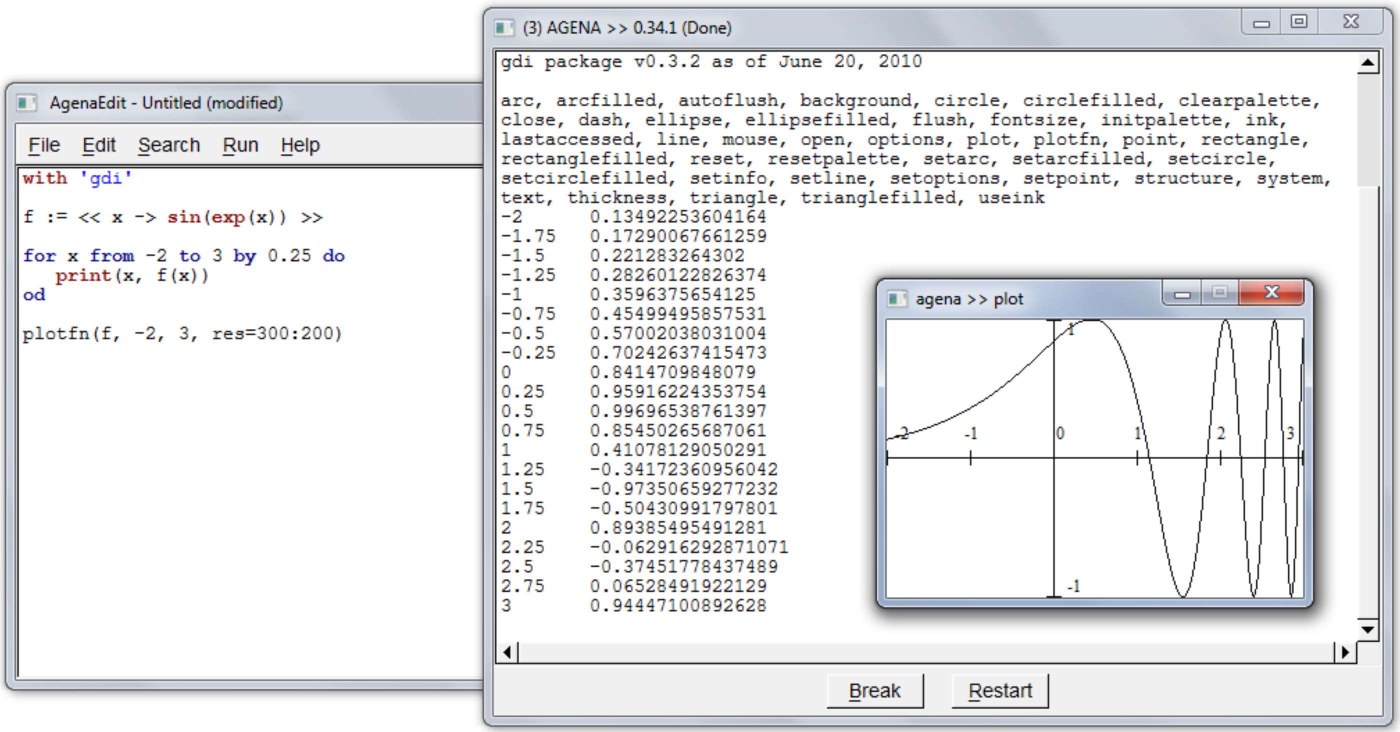
- In Solaris, Linux, Mac and DOS, type `agena` in a shell.



- 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 into a variable, type:

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

- Now the value 25 is stored into 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;
```

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 \ 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
```

- Exponentation with rational or integer power:

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

- Exponentation 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`

```
> 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]]]
```

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 tildes.

```
> 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):
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.

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 Statements & is 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 `is` operator checks a condition, too:

```
> result := is 1 < 2 then 'valid' else 'invalid' si;

> result:
valid
```

case Statements

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

```
> c := 'agena';

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

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;

> do
>     inc c
> as c < 10;

> c:
10
```

```
c := 0

> do
>     inc c
> until i = 10;

> c:
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

- 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
```

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'):
1.414213562373 1*I
```

Type Checking, 1

- You can check the type of arguments passed in two ways:
- Query the type with the `type` operator:

```
> f := proc(x) is
>    if type(x) <> number then error('wrong type of 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

- 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.

```
> 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
```

- This significantly speeds up recursively defined procedures.
- For the differences between `defaults` and `remember` check the manual (Chapter 7.24).

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 'progname.agn')` or starting Agena from the shell with `agena -i progname.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 := '% '
```

Miscellaneous

Precedence

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

<table>
<tr><th>Prio</th><th>Operators</th></tr>
<tr><td>10</td><td><code>or xor</code></td></tr>
<tr><td>9</td><td><code>and</code></td></tr>
<tr><td>8</td><td><code>< > <= >= = == <> :: :-</code></td></tr>
<tr><td>7</td><td><code>in subset xsubset union minus intersect atendof</code></td></tr>
<tr><td>6</td><td><code>& :</code></td></tr>
<tr><td>5</td><td><code>+ - split || ^^</code></td></tr>
<tr><td>4</td><td><code>* / % \ shift &&</code></td></tr>
<tr><td>3</td><td><code>not -</code></td></tr>
<tr><td>2</td><td><code>^ **</code></td></tr>
<tr><td>1</td><td><code>! ~~</code> and all other unary operators</td></tr>
</table>

Packages, 1

- Agena features various packages that can be invoked with the `with` or `readlib` functions, e.g. `with 'calc'`.

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
div	Fractions
environ	Access to the Agena environment
fractals	Various fractals & plotting routines, some FRACTINT support

Packages, 2

Package	Function
gdi	Graphics
gzip	Read and Write UNIX gzip compressed files
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
os	Functions to operate with the underlying operating system
rtable	Administration of remember tables
skycrane	Various easy-to-use wrappers to Agena functions

Packages, 3

Package	Function
stats	Statistical functions
strings	Various string handling functions
tables	Functions specialised on table processing
utils	Utility functions, e.g. CSV import and export
xbase	xBase file support (i.e. dBASE (tm) III+)
xml	XML decoding (LuaExpat)

Mathematical Constants

- Agena features the following numeric constants:

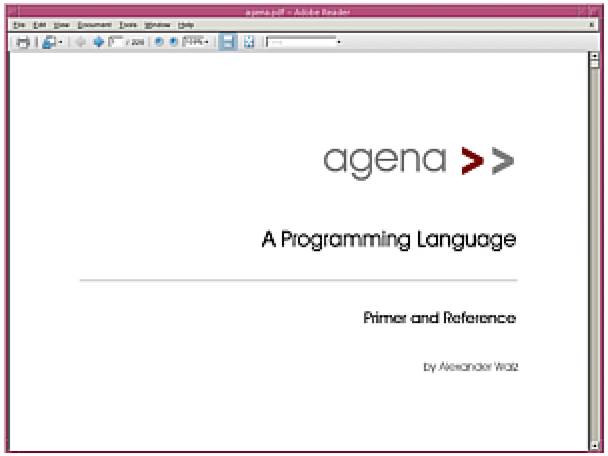
Constant	Meaning
`Eps`	Equals 1.4901161193847656e-08
`degrees`	Factor 1/Pi*180 to convert radians to degrees
`Exp`	Constant e = exp(1) = 2.7182818284590452354
`I`	Imaginary unit
`infinity`	Infinity
`Pi`	Equals 3.1415926535897932385
`radians`	Factor Pi/180 to convert degrees to radians
`undefined`	An expression stating that it is undefined, e.g. a singularity

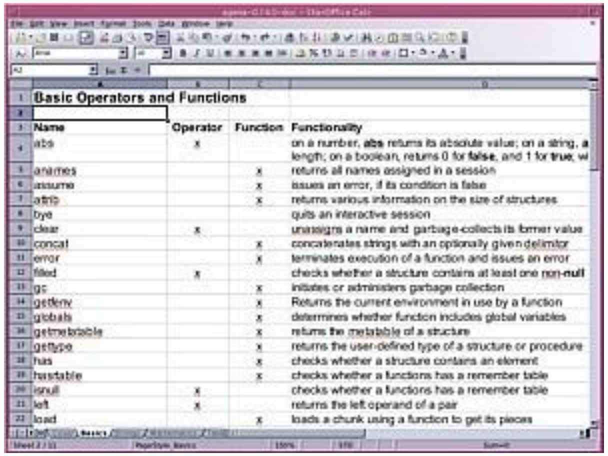
agena >>

Any Questions ?

- For further information, please consult
  - the *Primer and Reference*,
    a manual explaining Agena on 503 pages
  - the *Quick Reference*,
    an overview of all the functions available
- Both are available at

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





Basic Operators and Functions

Name	Operator	Function	Functionality
abs	x		on a number, **abs** returns its absolute value; on a string, a length; on a boolean, returns 0 for **false**, and 1 for **true**; wi
anames		x	returns all names assigned in a session
assume		x	issues an error, if its condition is false
attrib		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 delimitor
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
getfenv		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
hastable		x	checks whether a functions has a remember table
isnull	x		checks whether a functions has a remember table
left	x		returns the left operand of a pair
load		x	loads a chunk using a function to get its pieces