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, 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

Contents, 1

- Installing Agena
- Running Agena
- AgenaEdit
- 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
  - Write-Protection
- Control Statements
  - if Statements & if 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
  - Conditional for Loops
  - Loop Control
- Procedures
  - Procedures
  - Local Variables
  - Variable Number of Arguments
  - Options

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
  - Functional-Style Programming
  - with and Related Statements
  - Syntactic Sugar

Contents, 5

- Printing
- Did you know ?
- Miscellaneous
  - Precedence
  - Mathematical Operators
  - Mathematical Functions
  - Mathematical Constants
  - String Functions & Operators
  - Packages
- Excurse: Doing Math with Agena

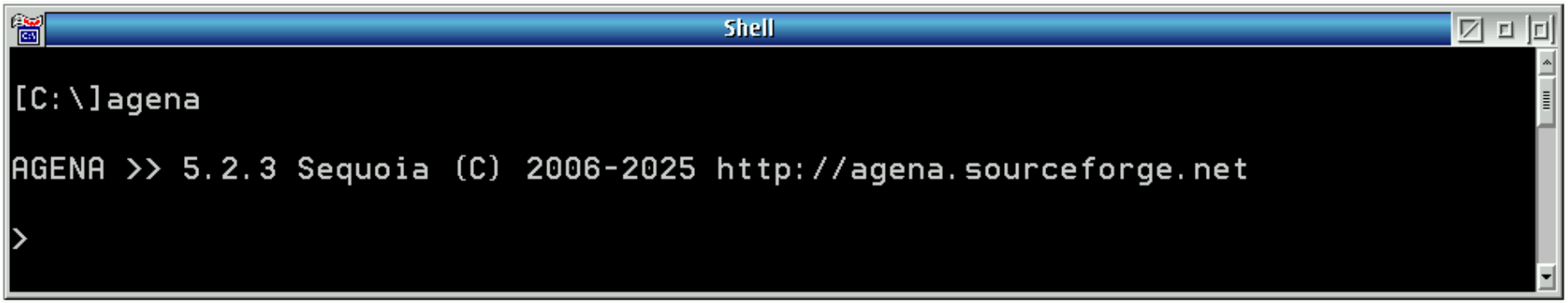
Getting Started

Installing Agena

- The installer automatically installs and sets up Agena on Solaris, OS/2, Linux, Windows, and Mac OS X.
- After installing the binaries, you don't need to change any settings.
- Information on how to install the DOS and Windows portable edition is included in the manual and the respective readme files.

Running Agena

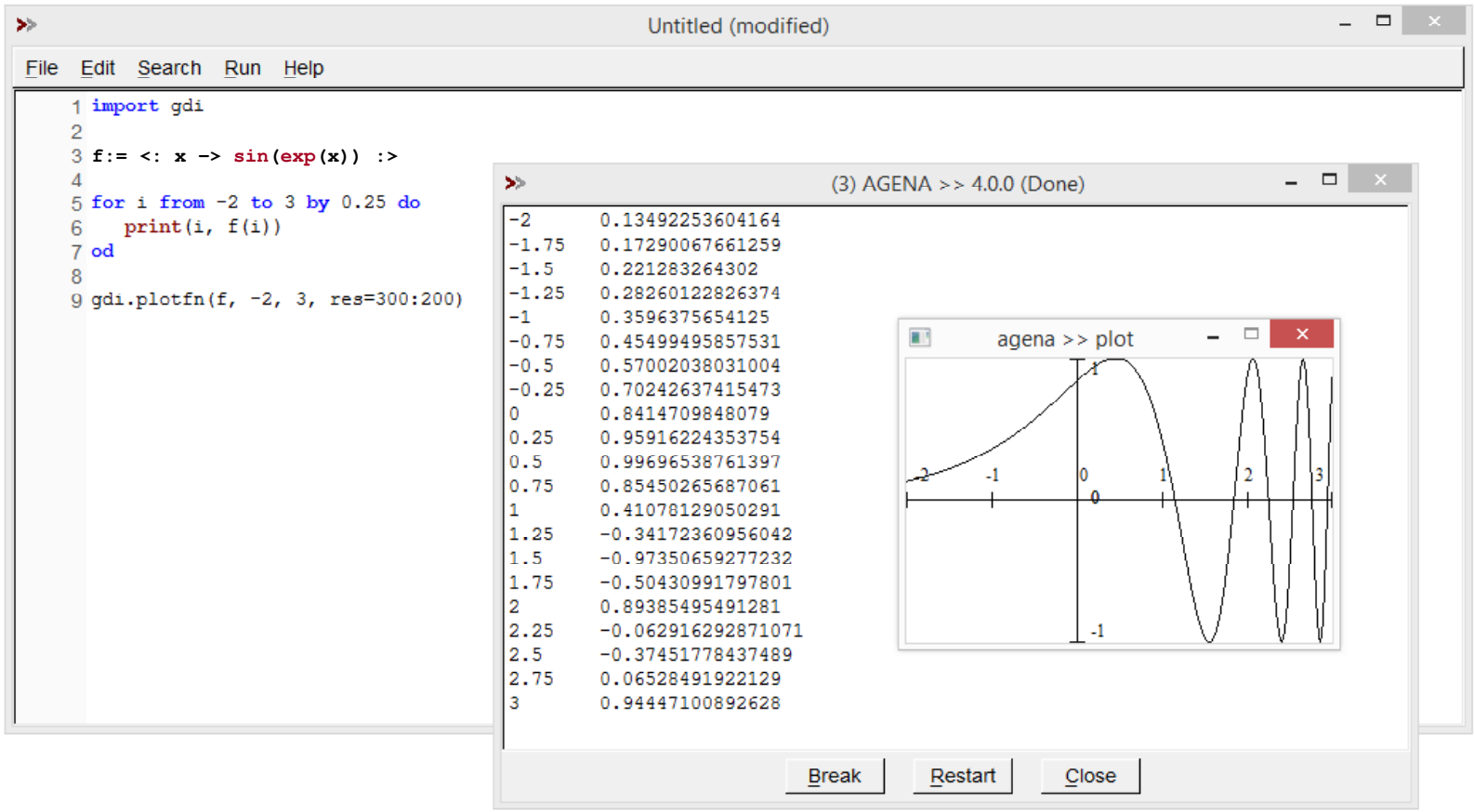
- In Windows and OS/2, simply click the >> icon in the programme group to start the interpreter.
- Or, on any operating system, just type `agena` in the shell.



- Enter statements at the '>' prompt.

AgenaEdit, 1

- AgenaEdit is an editor with syntax highlighting and a runtime environment for Windows, Solaris, Mac OS X and Linux. You can start it by typing `agenaedit` in a shell.



AgenaEdit, 2

- Type your programme in the editor window and press F5 to run it.
- To execute only specific lines, select consecutive lines in your programme with the mouse or keyboard, then press F6.
- To interrupt the current computation, press the Break button.
- To save or open your programs, use the File menu in the editor window.
- Browse the menu to explore all the features.

First Steps, 1

- You can enter any valid Agena code at the console, and how you end the line determines the output.
  - To evaluate an expression and print its value to the console, end the line with a colon (:).
  - If you want to evaluate an expression without printing its value, you can end the line with a semicolon (;) or nothing at all.
- You may optionally insert one or more white spaces between operands in your statements.
- To add the numbers 1 and 2 and see the result, simply type:

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

First Steps, 2

- To assign a value to a variable, type:

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

- The value 25 is now stored in the variable c. You can then use c to refer to this number in subsequent calculations.
- Suppose that c is 25° Celsius. If you want to convert to Fahrenheit, enter:

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

- The cls command clears the screen, restart clears all values and bye quits the interpreter. Use history.on() to enable command-line logging and history.off() to disable it.

Names & Assignment, 1

- 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 number in any order.
- Use the assignment operator := to assign a value to a name.

```
> a := 1;

> var := 'hello world';
```

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

```
> a := null;

> clear var;
```

Names & Assignment, 2

- Multiple variables can be assigned as follows:

```
> a, b, c := 1, 2, 3;

> a := 1, b := 2, c := 3
```

- The arrow operator assigns one and the same value to one or more names, and clear unassigns one or more names:

```
> a, b, c -> 1;

> clear a, b, c;
```

Names and Assignment, 3

- Compound assignment is supported in three ways:

```
> 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 >>

Names & Assignment, 4

- Further compound operators are:

Function	Compound
String concatenation	`&:=`
Bitwise AND	`&&:=`
Bitwise OR	`||:=`
Bitwise XOR	`^^:=`
Bitwise left-shift	`<<:=`
Bitwise right-shift	`>>:=`
Rotational left-shift	`<<<:=`
Rotational right-shift	`>>>:=`

Data Types

Integral & Rational Numbers

- You can represent numbers in the following ways:
- Integers:

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

- You can also print multiple values on a single 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 three notations to represent complex numbers.
- The ! operator (Cartesian notation), and the !! operator (polar form):

```
> 1!2, -1.1!-2, 3!0, sqrt(2)!!(Pi/4):
1+2*I   -1.1-2*I        3       1+I
```

- The I constant:

```
> 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 your 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 a rational or an integer power:

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

- Exponentiation with an integer power only (faster):

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

Strings, 1

- Strings can be enclosed in single or double quotes. Both are treated the same.

```
> '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
```

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
```

Boolean Expressions & Relations, 1

- Agena supports the Boolean values true, false, and fail. The constant fail is used to indicate 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

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

> 1 < 2 < 3:
true
```

Boolean Expressions & Relations, 3

- If you add, subtract, multiply or divide a Boolean - or an expression 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 complex data structures. They consist of zero or more key-value pairs, where the key references the position of the value in the table and the value contains the data itself.
- Tables can be nested:

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

long form

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

short form

- To get the data for key 1, type:

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

Tables, 2

- To get the second entry in the nested table, enter:

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

- You can create empty tables in two ways.

```
> 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.

Tables, 3

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

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

- When trying to index a null value with square brackets, Agena will return an error. When using curly brackets, however, Agena will return 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
```

Arrays

- Tables with positive integral keys are called arrays. You may pass an alternative initialiser for the keys:

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

> tbl := [from 0, 10, 11, 12]:
[0 ~ 10, 1 ~ 11, 2 ~ 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

- A dictionary is a type of table where the keys can be any kind of data, not just positive integers. You can enter key-value pairs in two ways:

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

- As with arrays, indexed names are used to access the corresponding values in 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, just index it as follows:

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

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

Sets, 2

- To add or delete items from a set, use the insert and delete statements.

```
> insert 'daisy' into s;

> delete 'daisy' from s;
```

- The in operator 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 read 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

- To delete items, either set their index position to null or apply the delete command.

```
> s[4] := null;

> delete [1, 2, 2] from s;
```

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

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

- When adding values, sequences are twice as fast than tables.

Registers, 1

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

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

- You can read the items the usual way:

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

- If a value has been deleted, the size of the register will 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 their respective operands. Standard indexing method using integers is also supported:

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

- To change a pair's left and right operands, do the following:

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

Write-Protection

- The freeze function write-protects a table, set, sequence, register, pair or userdata from modification.
- The unfreeze function removes the write-protection again.

Control Statements

if Statement, 1

- The if statement is used for checking conditions. It can optionally include elif and else clauses, but it must always be closed with fi.

```
> if 1 < 2 then
>    print('valid')
> elif 1 = 2 then
>    print('invalid')
> else
>    print('invalid, too')
> fi;
valid
```

- For a single statement without an else clause, a short form is available.

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

if Statement, 2

- You can use simple assignments in the conditions of if and elif statements. The then block will execute if the right-hand side of the assignment does not evaluate to false, fail, or null.
- Compare:

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

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

if Statement, 3

- The if clause allows you to combine assignments with the conditional check.

```
> if lnx := ln(1), lnx >= 0 then
>    print(lnx)
> fi;
0
```

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 with clause can precede the operator, allowing you to define one or more auxiliary variables that are local to that operator's scope 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
```

case Statements, 1

- The case statement compares values and facilitates the execution of associated 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
```

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 condition evaluates to true, the statements within the onsuccess clause will also execute.

```
> 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
```

Alternative end Keyword

- 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 is defined by an initial numeric value (from clause) and a terminal numeric value (inclusive, to clause). An optional step clause allows for custom incrementing. The loop body concludes with the od keyword.
- The current iteration value is stored in a control variable (e.g., i), which you can then use within 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 you omit the from clause, the loop will default to an initial value of 1.
- If the by 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.
- After the final iteration, the control variable is incremented by the step size one last time, so its value will be:

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

> i:
4
```


agena >>

for Loops, 4

- The 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 evaluates a condition prior to execution. The loop body will be repeatedly executed if this condition evaluates to true, or to any value other than false, fail, or null.
- 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
```

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
```

while Loops, 3

- You can combine an assignment and a conditional check in the while clause: When doing so, the assignment is redone each time flow control returns to the top of the loop, and the condition is checked again, as well.

```
> i := 0.3;
> while logn := ln(i), logn < -0.9 do
>    print(i, logn); i +:= 0.1
> od;
0.3     -1.2039728043259
0.4     -0.91629073187416
```

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

- do .. od loops are infinite 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

- The for loop will continue to iterate as long as the while condition evaluates to neither false, fail or null.

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

- The until condition similarly causes the loop to exit.

```
> for x to 10 until 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 x < 3;
1
2
3

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

Conditional for Loops

- This loop variant initialises a new local control variable, checks a while or until condition and then executes the loop body.
- You have to explicitly change the loop control variable in the body which allows for adaptive step sizes during a computation, for example when examining oscillatory functions with varying interval lengths.

```
> for i := 1 while i <= 3 do print(i); i +:= 1 od
1
2
3

> for i := 1 until i = 4 do print(i); i +:= 1 od
1
2
3
```

Loop Control, 1

- Agena features three statements to control loop execution. The following two are applicable to all loop types.
  - The next statement immediately starts a new iteration of the loop. This means any statements following the next keyword within the current loop iteration will be skipped.
  - Conversely, the break statement exits the loop entirely. Execution then continues with the first statement located immediately after the loop's conclusion.

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

Loop Control, 2

- next and break can also be combined with when or unless conditions:

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

> for i to 5 do
>    next unless i <> 3;
>    print(i);
>    break unless i <> 4
> od;
1
2
4
```

Loop Control, 3

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

```
> 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 completely restarts a for/to, for/in and conditional loop from its beginning.

```
> 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 replaced by the name used to call the procedure.

```
> 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, you can define a function using 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 define local procedures by placing the local keyword before the procedure keyword.

Local Variables

- A local variable is only accessible within the procedure and the specific block where it's declared. This means you can't use it in other procedures, at the interactive Agena level, or outside its declared block.

```
> 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 holds all variable arguments passed using the ? facility. You can access these values just like any other table. As a convenient shortcut, you can also use the question mark (?) directly to refer to varargs.
- The system variable nargs contains the number of arguments passed (regardless of whether you use the ? facility or not).

```
> f := proc(?) is
>    return nargs, ?, ?[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 will automatically be set to null at function invocation.

```
> 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.

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 with the second argument ‘domain’:‘real’.

```
> 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), xsqrt(-2, 'domain':'real'):
undefined   undefined
```

Options, 3

- If the left-hand operand of the pair in a function call denotes a string, you can spare the single quotes put around 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, too:

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

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

> success:
false
```

- Alternatively, the protect function traps errors.

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
```

- Give 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.
```

Type Checking, 2

- Up to five basic 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
```

- The return type 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.
```

Type Checking, 3

- 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.
```

Type Checking, 4

- Use the pre clause to perform additional checks on function arguments.

```
> 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.
```

Predefined Results

- Predefined results can be set with the rtable.defaults function which puts them into a remember table.
- Agena will return the given predefined result if it exists and does not execute the procedure body, so there is an increase in speed, too.

```
> 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 results if the rtable.remember function is run. 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.

State Tables

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

```
> add := proc(x) is
>    feature store;
>    if x = null then  # set default value zero
>       store[1] := 0
>    else
>       store[1] +:= x
>    fi;
>    return store[1]
> end;

> add():
0

> add(10):
10
```

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 calling a function this way, it has always the highest precedence.

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
```

Short-cut Procedures, 2

- The with clause allows to define local auxiliary 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

- You can implement methods for tables in an object-oriented 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

- Here is a constructor that creates 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: We're creating a new account class, derived from the existing one, that specifically prevents 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.

Functional-Style Programming, 1

- There are some functional-programming-style functions and operators that spare you some lines of code, see Chapter 6.32 in the Primer & Reference:

Function	Features
map, @	applies a function to each item in a structure
select, $	returns all the elements that satisfy a Boolean condition
remove	deletes all the elements that satisfy a Boolean condition
zip	combines two structures of equal size
addup	approximates series
mulup	computes products
foreach	generates structures
times	applies a function to intermediate results

Functional-Style Programming, 2

Function	Features
pipeline	applies one or more functions to a structure
reduce, fold	apply a function to each item of a structure or string and return an accumulated - that is summed-up - result
factory. curry	transforms a function with multiple arguments into a sequence of single-argument functions

with and Related Statements, 1

- The with statement unpacks table values, indexed by string keys, declares them local and then allows to access them in the block. After leaving the block, all the values listed 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
```

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]
```

with and Related Statements, 3

- Another useful variant allows you to easily define local variables for use within a specific code block.

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

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

Syntactic Sugar

- Just an overview of syntactic sugar:

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

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

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

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

```
> break unless x = 0;
> if x <> 0 then break fi;

> next unless x = 0;
> if x <> 0 then next fi;

> return unless x = 0;
> if x <> 0 then return fi;

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

Printing, 1

- The print function writes values - numbers, strings, Booleans, tables, etc. to the shell:

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

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

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

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

Printing, 2

- To print 10 decimal (fractional) places of sqrt(2), we put .10 in front of the f specifier:

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

- The %s formatter represents a string and %18.15f depicts a number with a total of 18 digits (pre-decimal places plus the decimal dot plus the fractional places), including 15 fractional places:

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

- For more information and examples, check Chapter 3.19 ‘Print Values’ in the Primer and Reference.

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

Did you know, 2 ?

- Data stored in CSV, XML and JSON files can be imported with the utils.readcsv, utils.readxml and json.decode 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.
- On some 64-bit flavours of Windows 2003 Server and Windows 2008 Server you may need to set the agena.exe binary file to Windows 2000 or Windows XP compatibility mode in order for the interpreter to start successfully.

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

Mathematical Operators

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
^	Exponentiation with rational power
**	Exponentiation with integral power
%	Modulus
\	Integer division

Mathematical Functions, 1

Function	Description
sin(x)	Sine
cos(x)	Cosine
tan(x)	Tangent
sec(x)	Secant
csc(x)	Cosecant
cot(x)	Cotangent
arcsin(x)	Inverse sine
arccos(x)	Inverse cosine
arctan(x)	Inverse tangent
sinh(x)	Hyperbolic sine
cosh(x)	Hyperbolic cosine

Mathematical Functions, 2

Function	Description
tanh(x)	Hyperbolic tangent
arcsinh(x)	Inverse hyperbolic sine
arccosh(x)	Inverse hyperbolic cosine
arctanh(x)	Inverse hyperbolic tangent
sinc(x)	Cardinal sine
cosc(x)	Cardinal cosine
tanc(x)	Cardinal tangent
exp(x)	Exponentiation e^x
ln(x)	Natural logarithm
log(x, b)	Logarithm of x to base b
sqrt(x)	Square root

Mathematical Functions, 3

Function	Description
cbrt(x)	Cubic root
root(x, n)	Non-principal n-th root of x
proot(x, n)	Principal n-th root of x
hypot(x, y)	Hypotenuse
gamma(x)	Gamma function
lngamma(x)	Logarithmic Gamma function
fact(n)	Factorial
erf(x)	Error function
abs(x)	Absolute value/magnitude
sign(x)	Sign
entier(x)	Rounds x downwards to the nearest integer

Mathematical Functions, 4

Function	Description
floor(x)	Rounds downwards to the nearest integer (same as entier)
ceil(x)	Rounds upwards to the nearest integer
int(x)	Rounds to the nearest integer towards zero
frac(x)	Fractional part
round(x, d)	Rounds x to d-th digit
even(x)	Checks for even number
odd(x)	Checks for odd number

Mathematical Constants

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

Constant	Meaning
`Eps`	Equals 1.4901161193847656e-08
`DoubleEps`	Equals 1.08420217248550443e-19
`degrees`	180/Pi to convert radians to degrees
`radians`	Factor Pi/180 to convert degrees to radians
`Pi`	Equals 3.14159265358979323846
`Phi`	Golden ratio (1 + sqrt(5))/2
`Exp`	Constant e = exp(1) = 2.71828182845904523536
`I`	Imaginary unit
`infinity`	Infinity
`undefined`	Yields an undefined result, like at a singularity

See also Chapter A3 of the Primer and Reference.

String Functions & Operators, 1

Function	Description
&	Concatenation operator
in	Searches for a substring
notin	Checks whether a substring is not included in a string
atendof	Checks whether a string ends in the given pattern
strings.find	Searches for a substring, supports pattern matching
strings.glob	Matches patterns including ? and * wildcards
strings.match	Looks for the match of a pattern
strings.hits	Number of occurrences of a substring pattern
strings.include	Inserts a substring
strings.replace	Replaces substrings
strings.remove	Removes a substring

String Functions & Operators, 2

Function	Description
size	String length
abs, char	ASCII code conversion
empty/filled	Checks for an empty/filled string
split	Splits a string into its words
strings.fields	Extracts given fields (columns) in a string
strings.trim	Removes leading and trailing white spaces
strings.ltrim strings.rtrim	Remove all leading/trailing white spaces or a given character or string
tonumber	Converts a string to a number
tostring	Converts a number to a string
strings.format	C-style formatting

String Functions & Operators, 3

Function	Description
strings.lower/ strings.upper	Converts to lower/upper case
strings.capitalise/ strings.uncapitalise	Capitalises/uncapitalises a string
strings.dleven	Damerau-Levenshtein distance of two strings
strings.fuzzy	Compares two strings case-insensitively and returns an estimate of their similarity
strings.jaro	Jaro(-Winkler) similarity of two strings
strings.shannon	Shannon entropy indicators
regex.new	Defines a regular expression pattern
regex.count	Counts the number of matches
regex.find	Searches a string with a regular expression

String Functions & Operators, 4

Function	Description
regex.match	Looks for the first match of a regex-pattern in a string

Packages, 1

- Agena features various packages.

Package	Function
ads	Database specialised on storing and retrieving strings
astro	Astronomical time and date functions
bags	Multisets, Cantor sets that count occurrences
bimaps	Bi-directional maps
binio	Functions for processing binary files
bloom	Bloom filter for strings and numbers
bytes	Bits and bytes twiddling
calc	Undergraduate Calculus package
clock	Functions to process hours, minutes and seconds
com	RS-232 communication via COM ports

Packages, 2

Package	Function
combinat	Combinatorial functions
convert	Physical unit conversion (lengths, weights, etc.)
cordic	CORDIC numeric functions
cuckoo	Cuckoo filter for strings
curses	(n)curses binding to build terminal applications
dd	Double-to-double arithmetic
dblhash	Number hash tables
div	Fractions
dual	Dual numbers, Automatic Differentiation
environ	Access to the Agena environment
fastmath	Numeric approximations

Packages, 3

Package	Function
fifo	First-in first-out queues
fractals	Various fractals & plotting routines, some FRACTINT support
fzy	Fuzzy string matching
gdi	Graphics
gmp	GNU Multiple Precision Arithmetic Library (GMP)
gzip	Read and write UNIX gzip compressed files
hashes	String and number hashes
heaps	Skewed & binary heaps plus AVL trees
iconv	GNU iconv port, to transform strings between codepages
ini	INI file encoding & decoding (iniparser binding)
ints	64-bit integer math

Packages, 4

Package	Function
io	Input/output functions for console and files
ival	Interval arithmetic
json	JSON encoding & decoding
kiss	Fast Fourier Transform
lifo	Last-in first-out queues
linalg	Undergraduate Linear Algebra
llist	Linked lists
long	80-Bit Floating-Point arithmetic
lookup	Lookup tables
maple	Aliases to Maple functions
mapm	Mathematical arbitrary precision library

Packages, 5

Package	Function
math	Additional mathematical functions
memfile	String memory files
minizip	Read and write ZIP compressed files
mpfr	GNU Multiple Precision Floating-Point Reliable Library (MPFR)
net	IPv4/IPv6-based data exchange over the Internet or LANs
numarray	Numeric C arrays
numcuckoo	Cuckoo filter for numbers, resizable
numfilter	Cuckoo filter for numbers, non-resizable
numtheory	Number Theory
os	Functions to operate with the underlying operating system
rbtree	Red-black binary trees

Packages, 6

Package	Function
regex	Regular expression matching (PCRE2)
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
sqlite	SQLite 3 database interface
stack	Functions for stack operations
stats	Statistical functions
strings	Various string handling functions
tables	Functions specialised on table processing

Packages, 7

Package	Function
tar	Functions to list, read and extract UNIX tar archives
trie	Prefix and suffix trees for strings
unimath	Common functions for floating-point data
utils	Utility functions, e.g. CSV import and export
xbase	dBASE (tm) III+ file support
xml	XML encoding & decoding (LuaExpat port)
zx	Sinclair ZX Spectrum numeric functions

Excurse: Doing Math with Agena

- There are four packages for undergraduate mathematics:
  - calc – calculus,
  - linalg – linear algebra,
  - stats – statistics,
  - combinat – combinatorics,
  - kiss – Fast Fourier Transform,
  - dual – Automatic Differentiation.
- With the exception of kiss, all these aforementioned packages are built-in, so you do not need the import statement initialise them.
- As you will see, you actually do not have to know much about the interpreter to do some undergraduate math.

Calculus, 1

- Define a function, for example f(x) = sin(x):

```
> f := <: x -> sin(x) :>
```

- Print a table of values, with and without formatting:

```
> for x from -1 to 1 by 0.5 do
>    print(x, f(x))
> od;
-1      -0.8414709848079
-0.5    -0.4794255386042
0       0
0.5     0.4794255386042
1       0.8414709848079
```

```
> for x from -0.5 to 0.5 by 0.25 do
>    printf('%+05.2f %+10.6f\n', x, f(x))
> od;
-0.50  -0.479426
-0.25  -0.247404
+0.00  +0.000000
+0.25  +0.247404
+0.50  +0.479426
```

- Determine all the zeros over [-5, 5]:

```
> calc.zeros(f, -5, 5):
seq(-3.1415926535898, 0, 3.1415926535898)
```

Calculus, 2

- Differentiate f at point 0:

```
> calc.differ(f, 0):
1
```

- Evaluate the third derivative of f at point 0:

```
> calc.differ(f, 0, deriv=3):
-1
```

- Compute the minimum and maximum values on the interval [-10, 10],

```
> calc.minimum(f, -10, 10):
seq(-7.8539816339745, -1.5707963267949, 4.7123889803847)

> calc.maximum(f, -10, 10):
seq(-4.7123889803847, 1.5707963267949, 7.8539816339745)
```

or try `calc.extrema(f, -10, 10)`.

Calculus, 3

- Determine points of inflection and saddle points:

```
> calc.inflect(f, 0, 4):
seq(0, 3.141592653589793))

> calc.saddles(<: x -> x^3 :>, -1, 1):
seq(0)
```

- Integrate f over [0, Pi]:

```
> calc.integ(f, 0, Pi):
2
```

- Compute the series `Sum(1/n!, n=0 .. 100)` to return an approximation of Euler's number:

```
> calc.fsum(<: n -> 1/fact(n) :>, 0, 100):
2.718281828459
```

Linear Algebra, 1

- Define two vectors in different fashions: In the simple form, just pass all components explicitly; or pass only the non-zero components:

```
> a := < 1, 2, 3 >:
< 1, 2, 3 >

> b := vector(3, [1 ~ 2]):
< 2, 0, 0 >
```

- Check whether a and b are parallel and have the same direction:

```
> abs(a+b) = abs(a) + abs(b):
false
```

- Set a vector component by indexing:

```
> b[3] := 1;
```

Linear Algebra, 2

- Now read the modified vector and its rightmost component - a negative integral index n depicts the |n|-th element from the right:

```
> b:
< 2, 0, 1 >

> b[3], b[-1]:
1       1
```

- Addition and subtraction:

```
> a + b:
< 3, 2, 4 >

> a - b:
< -1, 2, 2 >
```

Linear Algebra, 3

- Scalar, dot and cross product:

```
> 2 * a:
< 2, 4, 6 >

> a * a:
14

> linalg.crossprod(a, b):
< 2, 5, -4 >
```

- The determinant:

```
> linalg.det(A):
-59
```

Linear Algebra, 4

- Find the vector x which satisfies the matrix equation A x = b. The matrix constructor expects row vectors.

```
> A :=  < < 1, 2, -4 >, < 2, 1, 3 >, < -3, 1, 6 > >:
[  1, 2, -4 ]
[  2, 1,  3 ]
[ -3, 1,  6 ]

> b := < -6, 5, -2 >:
< -6, 5, -2 >

> linalg.linsolve(A, b):
< 2, -2, 1 >
```


agena >>

Statistics, 1

- First we define a distribution:

```
> s := seq(10, 8, 1, 6, 5, 2, 9, 7, 3, 4):
seq(10, 8, 1, 6, 5, 2, 9, 7, 3, 4)
```

- Minimum and maximum observations along with their position in the distribution:

```
> stats.min(s):
1       3

> stats.max(s):
10      1
```

- Arithmetic mean:

```
> stats.amean(s):
5.5
```

Statistics, 2

- The median:

```
> stats.median(s):
5.5
```

- For the first quartile, the median and the third quartile of a distribution, along with the minimum, the maximum observation and the arithmetic mean, in this order, enter:

```
> stats.fivenum(s):
seq(2.75, 5.5, 8.25, 1, 10, 5.5)
```

- Standard and absolute deviation:

```
> stats.sd(s):
2.872281323269

> stats.ad(s):
2.5
```

Statistics & Combinatorics

- Outliers:

```
> s := seq(-100, 8, 10, 1, 6, 5, 2, 9, 7, 3, 4, 1000):
seq(-100, 8, 10, 1, 6, 5, 2, 9, 7, 3, 4, 1000)

> stats.chauvenet(s):
seq(1000, -100)
```

- The Cartesian product:

```
> combinat.cartprod([[1, 2, 3], [30], [50, 100]]):
[[1, 30, 50], [1, 30, 100], [2, 30, 50], [2, 30, 100], [3, 30, 50],
[3, 30, 100]]
```

- Combinations and number of combinations:

```
> combinat.chosse(3, 2):
[[1, 2], [1, 3], [2, 3]]

> combinat.numbcomb(3, 2):
3
```

Combinatorics, 2

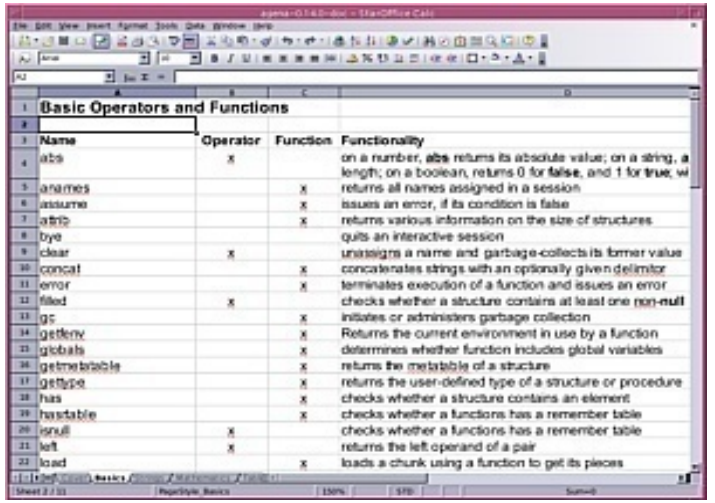
- Permutations and number of permutations:

```
> combinat.permute([1, 2, 3], 3):
[[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]

> combinat.numbperm(3, 2):
6
```

Any Questions ?

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



- Both are available at

http://sourceforge.net/projects/agena/files/Manuals