SCOL SERVER

Version 4

SCOL TUTORIAL LANGUAGE

Table of Contents

1. PRESENTATION 6
1.1. Presentation of the SCOL virtual machine 6
1.2. Link between the SCOL virtual machine and the files on your computer 6
1.3. Implementing the environment 7
1.3.1. Installing SCOL 7
1.3.2. Special configuration 7
1.3.3. Developing 8
2. HELLO WORLD 9
2.1. First version 9
2.2. Second version 11
2.3. Third version 11
3. PRINCIPLES OF SCOL PROGRAMMING 15
3.1. Environments and channels 15
3.1.1. Environments 15
3.1.2. Channels 16
3.1.3. Starting the SCOL machine: initial elements 16
3.2. Functional programming 16
3.3. Types and typing 17
3.3.1. Introduction to types 17
3.3.2. Syntax of SCOL types 18
3.3.3. Further information on types 19
3.4. Syntax of the SCOL language 20
3.5. Basic constructions 23
3.5.1. Main elements 23
3.5.2. Using tuples 25
3.5.3. Application to lists 26
3.5.4. Using tables 28
3.5.5. Further details on certain basic constructions 29
3.5.6. Using structures 29
3.5.7. Using type constructors 29
3.5.8. Using the functions 31
3.5.9. Redefining functions 32
3.5.10. New examples 33
3.5.11. Standard library 34
3.6. Global variables of the SCOL machine 40
4. CHANNELS AND COMMUNICATIONS 42
4.1. Manipulating channels 42
4.1.1. Channel manipulation API 42
4.1.2. Environment management API 42
4.1.3. Creating and destroying a channel 43
4.1.4. Creating and destroying a server 44
4.1.5. Additional functions of channel management 45
4.1.6. Script syntax 46
4.2. Communications in SCOL 46
4.2.1. Controlling connections: particular events 46
4.2.2. Sending a message using the _on function 47

4.2.3. Controlling message queues ... 48
4.2.4. Sending a UDP message ... 49
4.2.5. Another use for communication constructors ... 49
4.3. Methodology of network programming in SCOL ... 50

5. FILE MANAGEMENT ... 52

5.1. COL partitions ... 52
5.2. File types ... 53
5.2.1. Normal files ... 53
5.2.2. Signed files ... 53
5.3. File management API ... 54
5.4. Advanced file-reading functions ... 56
5.5. File selection interface ... 56

6. EVENT-DRIVEN AND GRAPHIC INTERFACE PROGRAMMING ... 57

6.1. Basic principles ... 57
6.1.1. Proprietary channel ... 57
6.1.2. Managing events ... 57
6.2. Examples ... 59
6.2.1. Windows ... 59
6.2.2. Timers ... 60

7. 3D PROGRAMMING ... 62

7.1. Basic 3D concepts ... 62
7.1.1. Scene ... 62
7.1.2. Session ... 63
7.1.3. Material ... 63
7.1.4. Performance ... 63
7.1.5. SCOL 3D Voy@ger characteristics ... 64
7.2. 3D file format ... 65
7.3. 3D manipulation API ... 68
7.3.1. New types ... 70
7.3.2. Session ... 70
7.3.3. General object management ... 71
7.3.4. Managing materials ... 73
7.3.5. Managing textures ... 75
7.3.6. Managing rendering and link with 2D interface ... 75
7.4. Managing collisions ... 76
7.4.1. Principles ... 76
7.4.2. API ... 78

8. BIGNUM PROGRAMMING ... 79

8.1. General introduction ... 79
8.2. API ... 79
8.3. Example ... 80

9. SQL ... 82

9.1. General introduction ... 82
9.2. API ... 82
9.2.1. Connection creation function: ... 82
9.2.2. Disconnection function: ... 82
9.2.3. Request function: ... 82

9.3. Examples **83**

10. HTTP INTERFACING **85**

10.1. Http server **85**
10.1.1. Principles 85
10.1.2. Some words of advice 86
10.2. Http client **86**
10.2.1. Principles 86
10.2.2. POST method 87

11. MULTIMEDIA PROGRAMMING **89**

11.1. RealPlayer API **89**
11.2. Quicktime API **89**
11.3. Basic multimedia API **89**
11.4. Audio API **89**
11.4.1. Mono recording/playback 90
11.4.2. Audio compression/decompression 90
11.4.3. Playback mixed with 3D sound 90
11.5. Video API **90**
11.6. Printing **90**

12. THE SCOL MACHINE: START-UP, CONTROL, STANDARD CLIENT AND SERVER **91**

12.1. SCOL Voy@ger: the supervisor **91**
12.2. Starting the supervisor **91**
12.3. Starting a SCOL machine with a start-up script **92**
12.3.1. Start-up script 92
12.3.2. Operating rights 93
12.3.3. Memory 93
12.4. Starting a machine or another process with a SCOL machine **94**
12.5. Communication between the SCOL machine and the supervisor **94**
12.6. Standard server and client **95**
12.6.1. General remarks on SCOL machine communication 95
12.6.2. Standard server – version 3 96

13. INTEGRATION POSSIBILITIES **98**

13.1. Interfacing via file exchange **98**
13.2. Interfacing via database: SQL library **98**
13.3. Interfacing via http: server or client http libraries **98**
13.4. Integration in a web page **98**
13.4.1. Simple interfacing 98
13.4.2. Sophisticated interfacing 101

14. DMS PROGRAMMING: DISTRIBUTED MODULES SYSTEM **104**

14.1. Presentation **104**
14.2. Definitions **105**
14.3. Principles **106**
14.3.1. Module architecture 106
14.3.2. Tree of documents 107
14.3.3. Encapsulation 107
14.3.4. Inter-module links and communication 107
14.3.5. Dynamic activation 108
14.3.6. Users and UserInstances 108

14.3.7. The SCS site editor 109
14.4. Downloading of resources 109
14.5. DMS site definition files 110
14.5.1. DMC file: distributed modules class 110
14.5.2. Dms files 112
14.6. 6 API 116
14.6.1. Server API 117
14.6.2. Client API 126
14.6.3. Editor API 131
14.7. Example 1 : module running only on the server. 132
14.8. Example 2: distributed module and zone management 136
14.9. Example 3: distributed module and intra-module message 138
14.10. C3d3 Module and plug-ins 141
14.10.1. Concept of the 3D Api 141
14.10.2. Anchors 148
14.10.3. Plug-ins 148
14.10.4. Examples 151

1. PRESENTATION

This chapter will help you take your very first steps in SCOL. It includes a presentation of the SCOL programming environment.

1.1. Presentation of the SCOL virtual machine

The SCOL virtual machine is the program based on SCOL technology. Its Windows version is called **UsmWin.exe** (which stands for 'Universal SCOL Machine for Windows'). You generally find it in 'C:/Program Files/SCOL/'.

As its name indicates, the virtual machine creates a virtual version of an 'ideal' machine whose characteristics would be the following:

- Automatic memory management: developers do not have to reserve or free up the memory themselves.
- This virtual machine's 'machine language' is the SCOL language.
- Fully integrated network communication management, since this management is masked by the SCOL language.
- A number of libraries, used to connect the machine to a simple graphic user interface, to sound interfaces and many others which we will look at later in this document.

As with any other machine, the SCOL virtual machine feeds on programs, written in 'machine language', in this case the SCOL language. This means that the virtual machine also includes a SCOL language compiler. The virtual machine is said to be 'universal' since it is the cornerstone of any system based on SCOL technology: such a system consists of a variable number of SCOL machines which communicate with one another.

Developing in SCOL amounts to writing programs in the SCOL language and in giving them to one or more SCOL machines for execution. It is important to note that more than one virtual SCOL machine can run on the same computer. Indeed, this will always be the case, since a special SCOL machine, called 'SCOL Voy@ger', will always be running as a background task on your system.

For those interested in the technical details, note that the SCOL language is compiled 'on-the-fly' towards a byte-code which is interpreted.

1.2. Link between the SCOL virtual machine and the files on your computer

As with any other machine, the SCOL machine needs mass memory to store the programs and data of the applications written in SCOL. The machine can therefore read and write files.

Specialists usually consider this to be a serious security problem: since SCOL applications are often connected to the Internet, the user does not want the content of his or her files to be transmitted somewhere else in the world. For this reason, SCOL's file management system has been specially designed to isolate those applications that do not use SCOL from those that do use it on the one hand, and to isolate the SCOL applications from one another on the other hand.

This system, which we will look at in further detail later on, is based on the concept of 'SCOL partition'. A SCOL partition is a directory (including all associated subdirectories) on your disk that can be used by SCOL: a SCOL machine cannot access a file not present in a SCOL partition from itself. A SCOL

machine can use a number of SCOL partitions, used in the order in which they are defined. A file which is not found in the first partition will be searched for in the second partition, and so on. Writing is always done in the first partition.

By default, the partitions on your SCOL machine running Windows are defined as follows:

- *c:/program files/scol/cache*
- c:/program files/scol/partition

When you develop your first programs, only the second will be used. The principle is as follows: the second partition is your working directory, while the first contains all the files you have gleaned while surfing from one site to another; consequently your files are protected from your wanderings.

The partitions are defined in the usm.ini file (whose path is usually c:/program files/scol/usm.ini). This file's syntax will be defined later in this document.

1.3. Implementing the environment

To become a 'real' SCOL developer, you must first install the development environment on your computer. To do so, proceed via the following stages.

1.3.1. Installing SCOL

Before doing anything else, you must install SCOL on your machine. To do so, download the latest version of the virtual machine from the Cryo-Networks Web site: www.cryo-networks.com. This program is free. It is approximately 1MB.

1.3.2. Special configuration

The SCOL machine can produce files that track its operation. These files are known as 'log' files. The creation of such files tends to slow down the machine. That is why they are not produced by default. However, these files contain information that is particularly useful for developers. In particular, they indicate syntax errors and, more generally, compilation errors. Moreover, certain SCOL language functions can be used to write directly in log files during execution. This often helps in the efficient debugging of your programs.

You must start by reactivating the log file creation function. Proceed as follows:

- Start SCOL
- Open the "**Settings**" menu
- Open the "**Expert Mode**" menu
- Replace the line `'echo 0'` with `'#echo 0'` (the `#` character is used to add comments to the line)
- Replace the line `'log no'` with `'log yes'`
- Click **OK**

You can also do this by editing the **usm.ini** file (usually located in *Windows* in *C:\program Files\SCOL*).

Your system is now configured and you are ready to develop your first program.

When you want to deactivate the log file creation function, simply restore these two lines to their original status.

1.3.3. Developing

To develop in SCOL, an editor is required for typing your programs.

A word of advice: use your favorite editor. Moreover, if this editor offers an automatic indent system for C language, activate it. The syntax of the SCOL language was designed to make the most of this widely used function.

If you don't have a favorite text editor, you can simply use the Windows **notepad**.

2. HELLO WORLD

We will start with a well-known program. It consists in displaying the message 'hello world' on-screen. We will give several versions of this program.

The aim of this chapter is to present several examples of programs written in SCOL. If some points seem obscure or insufficiently detailed, be patient. All will be revealed in the following chapter.

2.1. First version

To start with, create a 'Tutorial' directory in your 'SCOL/partition' directory.

Then type the file 'Tutorial/hello1.scol':

```
_load "Tutorial/hello1.pkg"
main
```

Now type the file '*Tutorial/hello1.pkg*':

```
/* commentary */
fun main()=
 _showconsole;
 _fooS ">>>>>>>>>>>> Hello World";;
```

Now double-click on the Tutorial/hello1.scol file to run the program.

A white, fine-lined window is displayed with a number of inscriptions that are not yet understandable. The last lines are:

```
exec : _load "Tutorial/hello1.pkg"

loading C:\Program Files\SCOL\partition\Tutorial/hello1.pkg ...
typechecking
fun main : fun [] S
bytecode producing
loading complete
exec : main
>>>>>>>>>>>> Hello World
WRITE 9
exec : _connected
_connected : command not found
WRITE_OK
>>end TimeOut
```

You see `'>>>>>>>>>>>> Hello World'` appear. This window is the **'console'** window and is closely linked to the log file since messages that pass in the console are written in the log files. Thus, once you have closed this console window, open the corresponding log file: it is a "`*.log`" file located by default in directory "`c:/program files/scol/log`", and contains the date (year-month-day_hour-minute-second). At the end of this file, you will find the replica of what is reproduced here. Well done! You have now created your first SCOL program. Now let's try to see what happened.

Generally speaking, the **.SCOL* files are start-up files called **script files**, whereas the **.pkg* files contain programs written in the SCOL language. Later on, we will see that there is a close tie between these two types of files. We will detail the syntax of the *script* files at a later stage.

The hello1.scol file contains the program start-up commands. The first line (`'_load ...'`) indicates that you must load the *hello1.pkg* program. The second line indicates that you must then run the '*main*' function.

The hello1.pkg file contains the program as such. In a SCOL program, you will find several types of definitions:

- functions, the definition of which starts with '`fun`'
- variables, the definition of which starts with '`var`' or 'typeof'
- new types, the definition of which starts with 't`ypedef`' or '`struct`'
- and still more things...

Each definition ends with a double semicolon '`;;`'. This is a reference to the Caml language developed by the French National Institute for Research in Computer Science and Control (INRIA).

Comments are placed as in C between `/*` and `*/`. Unlike C, you can interleave the comments, which often proves to be most practical.

In the `hello1.pkg` file, you thus define a '`main`' function. This function's name is followed by `()`: it does not take any arguments. The function's value follows the `=` sign. Each expression of the function is separated from the next by a semicolon (`';'`). Here, the function thus includes two expressions : `_showconsole` and `_fooS ">>>>>>>>>>>> Hello World"`

The definition of the function ends with a double semicolon.

The first expression calls the `_showconsole` function which displays the console window (which otherwise remains hidden).

The second expression calls the `_fooS` function which requires a character string-type argument, and displays it in the console window and, therefore, in the log file.

In SCOL language, arguments are not placed between parentheses nor separated by commas as in C language. The arguments follow the function, separated quite simply by spaces or new line characters. If you want to use parentheses, you can place them around one argument, or around the function+arguments. For example, the following three expressions are all correct:

```
_fooS (">>>>>>>>>>>> Hello World")
(_fooS ">>>>>>>>>>>> Hello World")
(_fooS (">>>>>>>>>>>> Hello World"))
```

We will come back to this important point later on. Note that if you don't like parentheses, you can use braces: they serve exactly the same purpose.

Once the console window is displayed, the virtual machine is always in operation. It only stops when you destroy the console window. Indeed, this is one of the ways of destroying a SCOL machine.

Towards the end of the log file (4 lines before 'hello world'), you will notice the following line:

```
fun main : fun [] S
```

This line is very important: it indicates that the virtual machine has successfully recorded (compiled) the main function, and gives its type: a function which takes no argument and which returns a character string. Indeed, with SCOL, you will very seldom have to define a function's type (number and type of arguments, type of the result) yourself. The SCOL machine does it for you (we talk of "inference of type").

2.2. Second version

file 'Tutorial/hello2.scol'

```
_load "Tutorial/hello2.pkg"
main "Test"
```

file 'Tutorial/hello2.pkg'

```
/* Hello2.pkg */

fun end(a,b,r)=
 _closemachine;;

fun main(title)=
 _DLGrflmessage
 (_DLGMessageBox _channel nil title "Hello World" 0)
 @end 0;;
```

This time, a simple dialog box is displayed instead of the console window.

This program underscores a number of points.

Firstly, the 'main' function uses an argument called 'title'. Here we can see how to pass this argument from the *hello2.scol* file. It takes the value "Test".

The 'main' function in the *hello2.pkg* file uses only one expression, but this expression calls a _DLGrflmessage function (which requires 3 arguments) whose first argument is itself the invocation of a _DLGMessageBox function (which requires 5 arguments). Without going into too much detail, let's say that the _DLGMessageBox function creates a dialog box whose title is the 3rd argument and whose text is the 4th. This function returns the identifier of the dialog box. The _DLGrflmessage function is used to define what will happen when the user closes the dialog box. The first argument is the dialog box, the second is the function to call. The '@' sign is used, as it were, to send a pointer to the 'end' function (to use the C language terminology). Without this sign, the 'end' function would be called immediately (which would provoke a compilation error since the 'end' function requires three arguments, whereas here only one is available).

The 'end' function contains a single expression that calls the '_closemachine' function, which closes the SCOL machine.

A quick look in the log file *(SCOL/hello2.scm.log)* reveals the following line:

```
fun main : fun [S] MessageBox
```

This time, the compiler has detected that the main function uses an S type argument (character string) and returns the MessageBox type. Note that the developer has not had to specify the argument type (S), but that the compiler has determined it itself.

2.3. Third version

file '*Tutorial/hello3.scol*':

```
_load "Tutorial/hello3.pkg"
main
```

file '*Tutorial/hello3.pkg*':

```
/* Hello3.pkg */

fun _end(a,b)=_closemachine;;

fun _resize(a,t,x,y)=_SIZEtext t x-2 y-2 1 1;;

fun main()=
 let _CRwindow _channel nil 150 150 400 300
  WN_MENU|WN_MINBOX|WN_SIZEBOX "Hello World"
 -> win in
 let _CRtext _channel win 1 1 398 298
    ET_VSCROLL|ET_HSCROLL "Hello World"
 -> text in
 (_CBwinDestroy win @_end nil;
  _CBwinSize win @_resize text
 );;
```

This program defines three functions: `_end`, `_resize` and `main`.

Here we see a number of new elements appear: `_SIZEtext`, `let`, `nil`, `_CRwindow`, `_CRtext`, `_CBwinDestroy` and `_CBwinSize`.

Functions `_SIZEtext`, `_CRwindow`, `_CRtext`, `_CBwinDestroy` and `_CBwinSize` are graphic interface functions:

`_CRwindow` is used to create a window:

- argument 1 is the window's proprietary channel (this will be explained later)
- argument 2 is the parent window of the window to create
- arguments 3 and 4 give the position of the window
- arguments 5 and 6 give the size of the window
- argument 7 contains the window's flags (here, you define a simple window that can be minimized and whose size can be changed using the mouse)
- argument 8 contains the window's title

`_CRtext` is used to create a text field:

- argument 1 is the text field's proprietary channel (this will be explained later)
- argument 2 is the parent window of the text field to create
- arguments 3 and 4 give the position of the text field
- arguments 5 and 6 give the size of the text field
- argument 7 contains the text field's flags (here, you define a text field with horizontal and vertical scroll bars)
- argument 8 contains the text placed initially in the text field

`_CBwinDestroy` defines the function to call when the window is destroyed:

- argument 1 is the window in question
- argument 2 is the function to call when the window is destroyed
- argument 3 is a user parameter

_CBwinSize defines the function to call when the window is resized:

- argument 1 is the window in question
- argument 2 is the function to call when the window is resized
- argument 3 is a user parameter

_SIZEtext resizes a text field:

- argument 1 is the text field to resize
- arguments 2 and 3 give the field's new dimensions
- arguments 4 and 5 give the field's new coordinates
-

Note the 'nil' function. In fact, nil is a special value which has no defined type. Any variable or parameter can take the value nil. Here, nil is used in two places:

- as the parent window of the window created by _CRwindow: this means that the window has no parent
- as the user parameter of the _CBwinDestroy function: this can mean that there is no user parameter to pass.

The 'let ... -> ... in ...' function is used to define local variables.

Between 'let' and '->', you can write any given expression. As with all expressions in SCOL, it returns a result (here, it is a window in the first instance, a text field in the second).

Between '->' and 'in', you write the local variable (here, 'win' and 'text').

After the 'in', you write the expression in which the local variable can be used. Beyond the expression that follows the 'in', the local variable is no longer recognized.

A quick word on callback functions: the functions passed as arguments of the _CBwinDestroy and _CBwinSize functions are callback functions, since they will subsequently be called when a given event occurs. By convention, callback functions use at least two arguments:

- \- the first argument gives the assigned object again (the window in the case of _CBwinDestroy, and _CBwinSize)
- \- the second argument is the user parameter as defined by the _CBwinDestroy or _CBwinSize function.

Possible additional arguments depend on the nature of the event. For example, the destruction callback does not use any additional parameters, while the resizing callback uses two additional parameters which contain the new size of the window.

We are now in a position to understand this third program.

The 'main' function creates a window and a text field contained in this window. The title of the window and the content of the text field are positioned on the 'Hello World' value. The 'main' function also defines two callbacks on the destruction and the resizing of the window.

If the window is destroyed, the program stops (_closemachine function).

If the window is resized, you must resize the text field. This is done by calling the _SIZEtext function. Note the use of the user parameter to 'pass' the text field to the callback.

We will use this little program in the next chapter, since it can be used to define a simple display area.

3. PRINCIPLES OF SCOL PROGRAMMING

3.1. Environments and channels

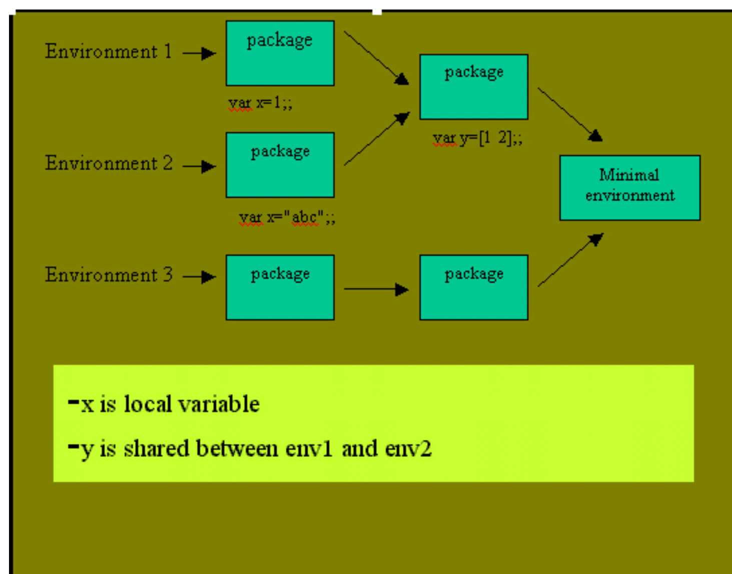
3.1.1. Environments

An **environment** is a list (in the computer sense of the word) of variables and functions. In the previous chapter's *'hello3'* example, in the SCOL machine there was an environment containing the `main, _resize and _end` functions and all the variables defined in the *locked/lib/const* file. Moreover, this environment contained all the SCOL API functions `(_CRwindow, _Crtext, ...)`.

Files written in SCOL are always compiled in a certain environment. The program thus compiled can refer to the environment's functions and variables; those defined in this file are added to the environment at the **start** of the list of variables and functions: by compiling the *'hello3.pkg'* file, we added the `main, _resize` and `_end` functions at the start of the environment. This means that any file compiled **afterwards** will be able to refer to these three functions.

By definition, a **minimal environment** is one that only contains the SCOL API.

The originality of the SCOL machine is that it can manage several environments at the same time: several lists of variables and functions coexist in the machine's memory. That is why it is important to know – at any time – in which environment you are working. These environments are not necessarily independent. As we said, an environment is a list. You can have two lists with the same end part. For example, lists (1,2,10,11,12) and (3,4,5,10,11,12) have the same size 3 end: (10,11,12). Similarly, in SCOL, two environments can have the same end. In fact, each environment ends with the **minimal environment**. This means you can share certain resources between the environments: functions and variables can be pooled. A variable that is **local to an environment** is one that is defined in a non-shared part of the environment's list.



3.1.2. Channels

A **channel** is a pair (environment, network connection). This association is an original feature of SCOL. At its minimum, the environment is the minimal environment. The network connection is usually a TCP/IP socket-type connection. However, you define a special type of channel, called **unplugged**, which does not contain any network connections.

Thus when you create a channel, you must specify its initial environment as well as the possible network connection. A channel's environment can develop over time:

- You can enlarge an environment by compiling new SCOL files.
- You can delete elements from an environment by removing the functions and variables located at the start of an environment.
- You can substitute one environment for another, and in particular reinitialize a channel by replacing its minimal environment.

3.1.3. Starting the SCOL machine: initial elements

When you start the SCOL machine, an `unplugged` channel is automatically created with the minimal environment. The `*.SCOL` file is used to define the operations to perform on this channel. These typically consist in compiling one or more SCOL files, then in starting the execution of a function. The files will be compiled in this first channel, and the function searched in its environment.

3.2. Functional programming

The SCOL language is a **functional** language, even if it authorizes imperative programming and all side effects.

The base unit of the SCOL language is the function: a function is a computer object which uses a certain number of arguments (possibly reduced to zero) and which produces a single result.

Any function returns a result, even when this result is not "significant." For example, in the *hello1* example, the `_showconsole` function returns a result (an integer). However, this result is not “important”; what the developer is interested in is the fact that the `_showconsole` function initiates the display of the console window. Any effect of a function other than the result of the function is called a **side effect.** A side effect must not make you lose sight of the fact that the function which produced it itself returned a result.

With its functional approach, SCOL encourages function interleaving. We have already seen one example of interleaving in the *hello2* program. Here is a second example. Others will follow.

```
fun f(x)= x+1;;
fun g(x)= x*2;;
fun gof(x)= g f x;;
```

Here, function `f` calculates what follows (`x+1`), function `g` calculates the double (`x*2`), function `gof` calculates the compound of `g` and `f` (`(x+1)*2`).

If we want to use parentheses for better readability, we could write:

```
fun gof(x)= g (f x);;
fun gof(x)= (g f x);;
fun gof(x)= (g (f (x)));;
```

However we could not write:

```
fun gof(x)= (g f) x;;
```

Other example:
```
fun f(a,b)=strcat a b;;
fun g(a)=strcat a ".";;
fun h(a,b)= f g a b;;
```

The `strcat` function is one which concatenates two character strings. For example (`strcat "a" "b"`) equals `"ab"`.

In this example, the `h` function takes two character strings and returns a string consisting of two arguments separated by a point.

We could write :
```
fun h(a,b)= f (g a) b;;
fun h(a,b)= (f (g a) b);;
```

But we could not write:
```
fun h(a,b)= f((g a) b);;
```

And especially not:
```
fun h(a,b)= f(g(a),b);;
```

Thus, the parentheses enclose the expressions, and not – as is the case with C – the list of a function's arguments. Since each argument is itself an expression, you can enclose an argument in parentheses. You can replace the parentheses with braces: they are completely equivalent.

3.3. Types and typing

3.3.1. Introduction to types

As was indicated in the *"Hello World"* examples, the SCOL machine automatically determines the types of the functions during the compilation. However, in some cases, developers will have to create certain types themselves: definition of structures, of type constructors, of certain variables and sometimes of function prototypes. Moreover, to be able to read the documents on the SCOL APIs, you need to be able to understand the types.

The **type** of an element in your program can be diverse: integer, character string, table, tuple, function... In SCOL, types are simply there to help the developer: the types are only used when compiling to detect most of the errors at that stage. Thereafter, when you run your program, they are no longer used since the program has already been proved to be correctly typed.

Typing is the operation whereby the types in your program are verified. This operation is performed at the same time as the compilation (it is said to be **static**). For example, it detects that you are using a character string with a function which requires an integer, but it can be much more subtle than that. When a type error is detected, the SCOL machine stops and displays an error message on the console window (and thus in the log file) explaining the nature and position of the type error.

In SCOL, typing is done by **inference of type**: this means that the compiler calculates the type of your functions itself; you will not usually have to specify it. SCOL typing is also said to be **polymorphic**. Certain functions do not impose any type conditions on certain arguments, or certain parts of certain arguments.

For example the following function:
```
fun f(x)=1;;
```

This function does not constrain the type of `x`, since this argument is not used. Function `'f'` is said to be polymorphic since it accepts arguments of different types.

This example is extremely basic, however there are more interesting ones: a function which calculates the size of a list will not usually be concerned with the type of the elements in the list. Rather than write one function for the list of integers, another for the list of character strings and so on, polymorphism allows you to only write one function.

3.3.2. Syntax of SCOL types

The SCOL language defines a syntax to write the types. This syntax is defined by the table below:

Type	=	*B* |	**u***n* |	**r***n*
	|	**tab** *Type* |	[*Type**] |	**fun** [*Type**] *Type*
TypeMono	=	*B* |	**r***n*	
	|	**tab** *TypeMono* |	[*TypeMono**] |	**fun** [*TypeMono**] *TypeMono*
B	=	Basic type		
u*n*	=	dependent variable		
r*n*	=	level *n* recursion		

The basic types are:

I	:	int
S	:	string
F	:	float
Chn	:	SCOL channel
Srv	:	SCOL server
Env	:	environment
Comm	:	communication

This list is not exhaustive: through type structures and constructors, you can develop your own basic types yourself.

Here are some comments on the table in case you are not familiar with this notation.

The first line defines the *Type* expression, which is actually the SCOL type. You then find the different ways of writing the expression, separated by '|': this can be:

- `B`, defined on the third line: it is a basic type, like `I, S, F,` ... Thus since I is a basic type, *B* can be written I, and since `Type` can be written `B, I` is a type in SCOL.
- **u***n* with integer *n*: `u0, u1, u2, u3, ...` are types: they correspond to the dependent variables (which we will detail at a later stage).
- **r***n* with integer *n*: `r0, r1, r2, r3, ...` are types : they define the recursions in the types (which we will detail at a later stage).
- **tab** `Type`: table type. The word **tab** is followed by the type of the elements in the table. For example, `tab I` is the type of a table of integers.
- **[***Type**]: tuple type. Between square brackets, you write several *Types* (represented by the asterisk *). For example, `[I S]` is a tuple of two elements, the first of which is an integer, and the second a character string. The tuple may be empty: `[ ]`. The tuple can itself contain tuples: `[I [S I]]`
- **fun [***Type**] *Type*: function type. The word fun is followed by a tuple containing the arguments of the function then the type of the result. For example '`fun [I I] S`' is a function which takes two integers as arguments, and returns a character string.

The expression *TypeMono* defines the monomorphic (non-polymorphic) types: the only difference with *Type* is the absence of dependent variables **u***n*.

3.3.3. Further information on types

3.3.3.1. Polymorphism

Let's take another look at the polymorphism example `fun f(x)=0;;` If you compile an example containing this function, you will see in the log file that the type detected by the compiler is: `fun [u0] I.`

`u0` represents 'Unknown 0'. This means that the type could not be determined and that it is immaterial. The 0 means you can differentiate the unknown types.

Other example: `fun f(x)=x;;`

This time the compiler will determine the following type: `fun [u0] u0`.

This means that the type of argument x is irrelevant, but that the type of the result is the same.

Other example: `fun f(x,y)=x;;`

This time the compiler will determine the following type: `fun [u0 u1] u0`.

This means that the type of the arguments is irrelevant, but that the type of the result is that of the first argument.

3.3.3.2. Recursion

We have seen that tuples can be interleaved `[I [S I]]`. What happens with an infinite interleaving?

Let's look at the concept of a list in SCOL. The *List* type does not exist in SCOL; by convention, you define lists as a tuple of two elements, the first of which is the first element in the list, the second the next in the list. `Nil` is the end of the list.

In SCOL, you will write the list of integers from 1 to 5 as follows: `1::2::3::4::5::nil`, or equally: `[1[2[3[4[5 nil]]]]]`.

The type of a list of integers should be: `[I[I[I[I ...]]]]`

Note `[I r1]` the type of this kind of list: tuple whose second element is a level 1 recursion.

Other example: take an alternating 'list': `[I[S[I[S ...]]]]`. This 'list' alternates integers and character strings. Note `[I[S r2]]` the type of this kind of list: tuple whose second element is a tuple whose second element is a level 2 recursion. Note: this is not a real list, since a real list will normally only contain elements of the same type.

Generally speaking, the only recursive types you will have to deal with will be lists, thus `[I r1]` for a list of integers, `[S r1]` for a list of character strings, etc. However, it is possible that you will create – whether by mistake or otherwise – functions whose type contains different recursion elements.

For those who want to take this matter further, you can represent a type by an oriented graph. The nodes are:

- the basic types (these are therefore end-nodes)
- the polymorphic types (these are therefore end-nodes)
- the tables: there is therefore 1 child which represents the type of the elements in the table
- the *n*-tuples: there are therefore *n* children each corresponding to one of the tuple's elements
- the functions: there are therefore 2 children, the first corresponding to the argument tuple, the second to the result

This graph actually looks like a tree. However, a branch can sometimes go from a node to a node which is itself, its parent, its grandparent, or others besides. This is where the recursion comes into action: r1 for a branch going to itself, r2 for a branch going to its parent, etc.

3.3.3.3. Some constraints

The `nil` constant has no defined type, or rather it has all the types at the same time.

A type is only valid if it does not contain any free variables: while "`u0`" and "`fun [ ] u0`" are syntactically correct, they are invalid.

Typically, if you write a function whose type returns a result containing a u*n* (for example `fun [ ] u0`), this means that your function returns nil in all cases. You must simply replace one `nil` by 0.

There is a restriction to polymorphism due to side effects: variables are necessarily monomorphic, whereas functions can be polymorphic.

3.4. Syntax of the SCOL language

Like any other programming language, SCOL uses a very precise syntax. The table below defines this syntax completely. If you are unfamiliar with this type of notation, refer to the previous chapter dealing with types: the syntax for types uses the same notation, but is simpler. Remember that the * indicate that the element is repeated a certain number of times (which may be zero). The braces leave the choice between several elements. For example, *{***I**,**S***}** corresponds to any given series of **I** and **S**. The characters in bold correspond to the syntax elements found in the source file of a SCOL program, while the elements in italics are rewrite elements whose meaning is given elsewhere in the table.

A file written in SCOL language simply contains the *SCOL* element as defined in the table. The *SCOL* element is simply a series of definitions (*Definition* in the table).

There are 8 sorts of definitions:

- `fun`: function definition
- `typeof`: definition of a variable by its type
- `var`: definition and initialization of a variable
- `struct`: definition of a type of structure
- `typedef`: definition of type constructors
- `defcom`: definition of a communication constructor
- `defcomvar`: definition of a variable communication constructor
- `proto`: definition of a function prototype

You can skip the following table for now, but it will no doubt come in useful later on.

SCOL	=	*Definition**
Definition	=	**fun** *Function* **(***Args***) =** *Program* ;;
	|	**typeof** *Var* **=***TypeMono* ;;
	|	**var** *Var* **=** *Val* ;;
	|	**struct** *NewType* **= [** *Fields* **]** *Function* ;;
	|	**typedef** *NewType* **=** *TypeConstr* ;;
	|	**defcom** *Com* **=** string *{***I**,**S***}** ;;
	|	**defcomvar** *Comvar* **=** *{***I**,**S***}** ;;

```
                | proto Function = Type ;;

Program    =    Expr                 |  Expr ; Program
Expr       =    Arithm               |  Arithm :: Expr
Arithm     =    A1                   |  A1 && Arithm       |  A1 || Arithm
A1         =    A2                   |  !A1
A2         =    A3                   |  A3 == A3           |  A3 != A3
           |    A3 < A3              |  A3 > A3            |  A3 <= A3
           |    A3 >= A3             |  A3 =. A3           |  A3 !=. A3
           |    A3 <. A3             |  A3 >. A3           |  A3 <=. A3
           |    A3 >=. A3
A3         =    A4                   |  A4 + A3            |  A4 - A3
           |    A4 +. A3             |  A4 -. A3
A4         =    A5                   |  A5 * A4            |  A5 / A4
           |    A5 *. A4             |  A5 /. A4
A5         =    A6                   |  A6 & A5            |  A6 | A5
           |    A6 ^A5               |  A6 << A5           |  A6 >> A5
A6         =    Term                 |  -A6                |  ~A6
Term       =    ( Program )          |  ( Program ;)
           |    { Program }          |  { Program ;}
           |    int                  |  'char              |  nil
           |    string               |  [Arithm* ]

           |    Var(.Term)*          |set Var(.Term)* = Arithm
           |    Var(.NameOfField)*   |set Var(.NameOfField)* = Arithm

           |    Function  Args_Function  |  @Function

           |         let Arithm -> Locals in Arithm
           |         if Arithm then Arithm else Arithm
           |         while Arithm do Arithm
           |         mutate Arithm <-[ {_ , Arithm}* ]
           |         exec Arithm with Arithm

           |    Constr  Arithm       |  Constr0            |match Arithm with Case

Args_F     =    Arithm ...Arithm : as many Arithms as the F function has arguments
Args       =    nothing              |  Args'
Args'      =    Local                |  Local , Args'
Locals     =    Local                |  [ Locals' ]
Locals'    =    {_ , Locals}*

Val        =    Val'                 |  Val' :: Val
Val'       =    int                  |  'char              |  nil
           |    string               |  - int              |  [ Val* ]
           |    (Val)

Fields     =    Field                |  Field, Fields
Field      =NameOfField : TypeMono

TypeConstr =    TypeConstr'          |TypeConstr' | TypeConstr
TypeConstr'=    Constr  TypeMono     |  Constr0
```

Case = *Case'* | *Case'* | *Case* | **(** _ **->** *Arithm***)**
Case' =(*Constr* *Local* **->** *Arithm* **)** | **(***Constr0* **->** *Arithm* **)**

- *`Var`* = name of variable
- *`Function`* = name of function
- *`NewType`* = new basic type defined by the developer
- *`Local`* = local variable (dependent)
- *`NameOfField`* = name of field in a structure
- *`Constr`* = type constructor
- *`Constr0`* = empty type constructor
- *`Com`* = communication constructor
- *`Comvar`* = variable communication constructor
- `int` = integer
- `char` = character
- `string` = string

The integers can be coded in the following bases:

- decimal : 12349
- hexadecimal : 0x3fe
- binary : 0b10011
- octal : 0o234235

They are coded on 31 signed bits.

The chars are used to retrieve a character's ASCII code: 'A is an integer equal to 65.

The character strings are written between quotes. The \ character is used to access certain commands:

- \n : carriage return
- \z : NULL character
- \" : quote
- \\ : \
- \decimal number : \132 is the character 132

A \ at the end of the line tells the compiler to disregard the new line.

As in C, comments are written between `/*...*/` and can be interleaved.

The (*Program*), `(`*`Program`*` ;)`, `{`*`Program`*`}` and `{`*`Program`*` ;}` constructions are all equivalent.

3.5. Basic constructions

3.5.1. Main elements

3.5.1.1. Variables

You can define a variable using `typeof`. For example:

```
typeof x=[I I];;
```

The `x` variable is created, it is a two-integer tuple. The variable is initialized to `nil`.

If you want to directly assign it a value, use `var`:

```
var x=[1 2];;
```

However, `var` cannot be used for certain operations, in particular arithmetic operations.

You can modify the value of a variable using the construction: **set** *Var* **=** *Arithm*

For example: `set x=a+b`

This produces a side effect: it is not the result of the set function you are interested in, it is the fact that the value of x has changed.

3.5.1.2. Functions

To define a function, place `fun` in front of the name of the function followed by the list of arguments between parentheses and separated by commas. Then the '`=`' sign precedes the body of the function which ends with a double semicolon. For example, the '`sum`' function:

```
fun sum(x,y)=x+y;;
```

The typing will determine the type of the function: `fun [I I] I`

Here the function takes two integers and returns one integer.

You can write arithmetic expressions as standard: `(x+y)*z/w`. You can also use logical C operators: `&, |, ^, <<, >>, ~, &&, ||, !, ==, !=, <, >, <=, >=`.

Arithmetic operations and comparison operations are also available for floating numbers (type F), however you must place a point after the operator: `+., -., >., ...`

3.5.1.3. Tests and conditions

One of the most important constructions in programming is the conditional construction, i.e., the traditional '`if ... then ... else ...`'. This construction exists in SCOL in the same format:

```
if C then T else F
```

where `C` is a condition, `T` (respectively `F`) is the expression to perform if the condition returns true (respectively false).

In SCOL, you must always define the '`else`' expression.

The condition is necessarily an expression that returns an integer. The result of the condition is considered to be "true" if the integer is other than zero, and false if it equals 0. In the condition expression, you can use C Boolean operators: `&&` and `||`. As in C, the expression is not necessarily completely evaluated:

- `A && B` : if A is false, B is not evaluated (the result is inevitably false)
- `A || B` : if A is true, B is not evaluated (the result is inevitably true)

The `T` and `F` expressions must return the same type, i.e. the type returned by the construction.

Indeed, the `'if ... then ... else ...'` construction is a function which returns the value of T or of F according to the value of the condition. The type of this function is:

```
fun [I u0 u0] u0
```

You can integrate the construction in an expression:

```
1+if x==0 then 1 else 3
```

This expression would equal 2 if x equals 0, 4 otherwise.

The same applies for the following function:

```
fun f(x)=if x then "a" else "b";;
```

The `'f'` function returns the `"a"` string if x is non-null, `"b"` otherwise. In imperative languages such as C, you do not have this type of operation.

3.5.1.4. Example

To allow you to carry out your tests, we will modify the hello3 program presented in a previous chapter.

file '*Tutorial/mytest.SCOL*:

```
_load "Tutorial/mytest.pkg"
main
```

file '*Tutorial/mytest.pkg*':

```
/* MyTest.pkg */

/* My tests */
fun sum(x,y)=x+y;;

fun mymain()=
 itoa sum 10 30;;

/* Common part */

fun _end(a,b)=_closemachine;;

fun _resize(a,t,x,y)=_SIZEtext t x-2 y-2 1 1;;

fun main()=
 let _CRwindow _channel nil 150 150 400 300
  WN_MENU|WN_MINBOX|WN_SIZEBOX "My Test"
 -> win in
 let _CRtext _channel win 1 1 398 298
    ET_VSCROLL|ET_HSCROLL ""
```

```
 -> text in
 (_CBwinDestroy win @_end nil;
  _CBwinSize win @_resize text;
  _SETtext text mymain
 );;
```

In this example, you simply display a text window in which you write (function `_SETtext`) the result of the 'mymain' function.

The 'mymain' function must always return a character string. If you are using integers, use the 'itoa' function to convert them into character strings before returning them. Using 'itoa' and 'strcat' (which concatenates two character strings), you can display all the results.

Here we are testing the 'sum' function on two integers 10 and 30.

In the examples that follow, we will simply give the functions that are to be inserted in place of 'mymain' and 'sum'. You can carry out your tests yourself.

3.5.2. Using tuples

Tuples are a very practical way of handling heterogeneous sets of data. Their use hides an automatic memory allocation; that is why tuples do not exist in C. Only a language that automatically manages the allocation and deallocation of the memory can really use tuples.

The term [*Arithm**] constructs a tuple from different expressions present between square brackets. For example:

```
[ 1 2  nil [ 3 4 ] ]
```

is a size 4 tuple, whose last element is a size 2 tuple.

You retrieve a tuple's components using the `let` function:

```
let [1 2 nil [ 3 4 ] ] -> [ a _ b c ] in A
```

In A, the `a`, `b` and `c` variables equal `1`, `nil` and `[ 3 4 ]` respectively. You 'skip' a value of the tuple by inserting an underscore character '_'. This means that you do not want to use this value, nor redefine a local variable to receive it.

The `mutate` function modifies one or more fields of a tuple:

let [1 2 nil [3 4]] -> tupletest **in**

mutate tupletest <- [_ 5 6 _]

In this example, you start by creating a tuple called `tupletest`. The 'mutate' function replaces the values `2` and `nil` of the tuple with 5 and 6 respectively, without affecting the other fields (where you have inserted an underscore '_'). This function is 'dangerous' since it modifies all the variables that refer directly or indirectly to the tuple. It is better to recreate a tuple.

Example: we will create a size 3 tuple, modify the second element using `mutate` and display its status before and after the modification.

```
/* My tests */
fun Tuple3toStr(t)= let t->[a b c] in
 strcat strcat strcat strcat strcat strcat
  "[" itoa a " " itoa b " " itoa c "]\n";;

fun mymain()=
 let [1 2 3]-> mytuple in
 strcat Tuple3toStr mytuple
```

```
(mutate mytuple <- [_ 10 _];
 Tuple3toStr mytuple);;
```

You will notice that the value of the expression **(***Arithm* **;** *Arithm***)** is the value of the last *Arithm* term. This means that the result of the first *Arithm* term is lost. This imperative approach (non-functional) is usually due to a side effect: here, the 'mutate' function creates the side effect. You can thus see the disadvantage of this approach: the result of the first term is lost, which means that information has been lost. This information has therefore not been verified during the typing, which renders the program more fragile.

When you run this program, you obtain the following result on the text window:

```
[1 2 3]
[1 10 3]
```

3.5.3. Application to lists

As we mentioned previously, lists in SCOL are managed in the form of size 2 tuples. The first element of the tuple is the first element in the list, the second element of the tuple is the next element in the list. For example, let `l` be a list:

```
let l->[val next] in ...
```

This expression is used to respectively retrieve the first element in the list and the next element in the list in the `val` and `next` local variables. The function which returns the first element in the list is called `hd`. The function which returns the next element in the list is called `tl`. These functions are present in the SCOL language, however you can write them in SCOL as follows:

```
fun hd(l)= let l->[val _] in val;;
fun tl(l)= let l->[_ next] in next;;
```

The type of these functions is:

```
hd : fun [[u0 r1]] u0
tl : fun [[u0 r1]] [u0 r1]
```

Here, you will notice that the two functions are polymorphic: they can be used for any list.

The list always finishes with the empty list, equal to `nil`.

The easiest way to build a list is to use the '`::`' list builder. For example, the expression `1::2::3::4::5::nil` builds a list of the first five integers. The type of the '`::`' function is:

```
fun [u0 [u0 r1]] [u0 r1]
```

Lists are a very important element in terms of functional languages because they are data structures of unlimited size (the automatic memory management facilitates their use). They are also data structures which go well with recursive processing.

Let us take a closer look at the lists and study the following examples:

3.5.3.1. Size of a list

In the following example, the `mysizelist` function calculates the size of a list. This function is recursive:

- the `nil` list has a size of 0
- any non-empty list has a size of 1 + the size of the list without its first element

The `mymain` function applies the `mysizelist` function on a list of 5 elements.

The program displays the number 5 in the text window.

```
/* My tests */
fun mysizelist(l)=
 if l==nil then 0
 else let l -> [_ next] in 1 + mysizelist next;;

fun mymain()=
 itoa mysizelist 1::2::3::4::5::nil;;
```

The type of the `mysizelist` function is naturally polymorphic:

```
mysizelist : fun [[u0 r1]] I
```

In actual fact, the `sizelist` function is already defined in the language: you do not therefore have to rewrite it yourself, but it offers a good example.

3.5.3.2. 5.3.2 Quicksort on integers

A classic example: **Quicksort**.

For this we define three functions, plus a fourth used to display the result.

The `conc` function takes two lists `p` and `q` and returns a new list which concatenates both lists in this order. It is a recursive function:

- if p is the empty list, the concatenation of `p` and `q` equals `q`
- otherwise the concatenation of lists `p` and `q` is a list in which:
  - the first element is the first element in list `p`
  - the rest of the list is the concatenation of list `p` without its first element or list `q` (recursion).

The `dividelist` function is a little more complicated: its role is to divide a list into two sublists according to an integer called the 'pivot'. The elements in the list are placed in one of the two sublists depending on whether they are less than or greater than the pivot. Here is an important tip: the `dividelist` function returns two lists, where normally a function can only return a single result. The solution is to use tuples: the `dividelist` function returns a tuple with two elements which are the two sublists. Thus, in SCOL, tuples are basically used for two purposes: to manage lists, and to group elements together to form a single element, which is easier to manage.

The `dividelist` function is not polymorphic, since it supposes (through the use of the `>` function) that it is processing a list of integers.

The `quicksort` function thus works on the basis of a very simple recursive principle:

- - the sorted empty list is always the empty list
- - if the list is not empty, by taking the first element in the list for the pivot, the sorted list is the concatenation of:
  - the sorted list of elements less than the pivot
  - the pivot
  - the sorted list of elements greater than the pivot

The `quicksort` function is not polymorphic since it calls the `dividelist` function.

The `display` function is simply used to create a character string representing the list, where the elements are separated by `'::'` and which ends with `'nil'`. This function is not polymorphic, since it supposes (through the use of the `'itoa'` function) that it is processing a list of integers.

The result of this program is: `2::3::5::6::8::nil`

Later on, we will see a quicksort variant which works on ordinary lists.

```
/* MyTest */

/* concatenation */
fun conc(p,q)=
 if p==nil then q
 else
  let p -> [a n]
  in a::conc n q;;

/* quicksort */

fun dividelist (x,p)=
 if p==nil then [nil nil]
 else
  let p->[a n] in
  let dividelist x n ->[r1 r2] in
  if x>a then [a::r1 r2]
  else [r1 a::r2];;

fun quicksort (l)=
 if l==nil then nil
 else
  let l->[vl nl]
  in let dividelist vl nl -> [va na]
     in conc quicksort va vl::quicksort na;;

/* display list */
fun display(l)=
 if l==nil then "nil"
 else
  let l->[a n]
  in strcat strcat (itoa a) "::" display n;;

fun mymain()=
 display quicksort 3::5::2::8::6::nil;;
```

3.5.4. Using tables

Tables are rarely used in a functional language. Lists are used as a preference for three reasons:

- tables are less suitable for recursive algorithms.
- tables are not expandable; you define their size once only.
- you cannot prevent the developer from writing a program that attempts to use a cell in the table outside its limits. This index overflow cannot be detected during the compilation, and is thus a source of error.

However, the table holds one advantage over the list: elements are all accessed in constant time.

A table can in particular be created using the **mktab** command, which takes the size of the table and an initialization value as arguments.

The i-th element in the table T is accessed by writing: `T.i`

If the table T is a table of tables, the j-th element of the i-th element of T is accessed by writing: `T.i.j` (indexes can be accumulated without limit).

To modify a value in the table, you simply write: `set T.i = ...`

3.5.5. Further details on certain basic constructions

The " **set** X = V " function returns the value V (the storage of V in X is merely a side effect).

The " **while** Condition **do** Expression " function calculates the condition (which must be an integer). If this integer is true, the expression is calculated and the condition evaluated again until it is false. It returns the result of the last expression calculated (nil if no expression was calculated, i.e., if the condition was false right from the first evaluation).

The " **let** X -> N **in** Y " function calculates X, creates the local variables contained in N then calculates Y and returns the result of Y. The static scope of variables contained in N is limited to expression Y.

3.5.6. Using structures

Structures are used a bit like in C. A structure is a particular type containing one or more fields. Each field is defined by a *field name* and an associated type. For example, a structure Rec containing three integers and a string is written:

```
struct Rec = [xRec:I,yRec:I,zRec:I,nameRec:S] mkRec;;
```

In this example, the developer chooses the names Rec, xRec, yRec, zRec, nameRec and mkRec. The only constraint is that the name of the new type (here Rec) must begin with an uppercase letter.

Where x is an object of type Rec, the different fields are accessed by writing x.xRec, x.yRec, x.zRec or x.nameRec. The names of the fields are considered as functions: for example, xRec is a function of type " fun [Rec] I ". For this reason, if two structures use the same field name, there will be an overlap: the second definition will hide the first.

To build an object of type Rec, you need a constructor: this is the role of mkRec which, in this case, is a function of type " fun [[I I I S]] Rec ". Example:

```
fun main()=
 let mkRec [1 2 3 "abc"] -> r
      in r.xRec;;
```

This function returns 1.

To modify the value of a field, simply write:

```
      set r.xRec = ...
```

3.5.7. Using type constructors

In some cases, it can be useful to use a variable with several different types: in variable x, you sometimes want an integer, other times a character string, and other times a tuple. Typing forbids this type of operation. To make it possible, you must use the equivalent of the *union* in C: these are type constructors. Take an example:

```
typedef U =
   xU I
 | sU S
 | tU [I I]
 | nU ;;
```

This defines a new type U, which can contain either an integer, a string, a two-integer tuple or nothing at all. The names xU, sU, tU and nU are called **type constructors**. They are considered as functions. For example, '`xU`' is a function of type " `fun [ I ] U` ".

We refer to them as constructors since only they can be used to construct an object of type U. The object constructed in this way contains two pieces of information: the name of the constructor used, and the useful value. The last nU constructor is special since it does not use a value; it is called **constructor0**.

An object of type U is processed using the '`match`' function.

```
fun numconstr(x)=
match x with
 (xU u -> 0)
|(sU v -> 1)
|(tU [u v] ->2)
|(nU -> 3);;
```

This function takes an element x of type U, and returns `0, 1, 2` or `3` respectively for `x` constructed with `xU, sU, tU` or `nU`. You can define one line of cases by default:

```
fun from_sU(x)=
  match x with
 (sU u -> 1)
|(_ -> 0);;
```

This function takes an element x of type U and returns 1 if x was built with `sU`, 0 in all other cases.

If the program does not define any cases by default `(_->…)`, the compiler "adds" the line `(_->nil)`.

The match function is not content with merely finding a variable's constructor, it also retrieves the value of the construction, as illustrated in the following example:

```
/* MyTest */

typedef Node =
   Int I
 | Add [Node Node]
 | Mul [Node Node];;

fun EvalNode(n)=
 match n with
   (Int x -> x)
  |(Add [a b] -> (EvalNode a)+(EvalNode b))
  |(Mul [a b] -> (EvalNode a)*(EvalNode b));;

fun mymain()=
 itoa EvalNode Mul [Add [Int 1 Int 2] Int 3];;
```

In this example you define a Node type used to code trees of expressions containing integer constants, additions and multiplications. The '`EvalNode`' function calculates the values of such a tree.

Here, the `mymain' function calculates the tree corresponding to the expression: (1+2)*3

3.5.8. Using the functions

SCOL can handle functions in the same way as integers or character strings. To do so we use language commands.

The '@' operator is used to convert a function name to a function object. Thus, the compiler considers that a function name not preceded by '@' represents a call to the function, with the parameters that follow. When preceded by '@', the compiler considers that a function object must be created for a subsequent operation.

In order for the handling of functions to be useful, you need to be able to apply a function object to a set of arguments and calculate the result. For this purpose you use the **exec...with...** function. Example:

```
fun baradd(x,y)= x+y;;
fun barmul(x,y)= x*y;;

fun foo(x,y,f)= exec f with [x y];;

fun main1()= foo 1 2 @baradd;;

fun main2()= exec @baradd with [1 2];;

fun main3()= baradd 1 2;;
```

The three `main1, main2` and `main3` functions return the same result.

The type of the foo function is noteworthy:

```
foo : fun [u0 u1 fun [u0 u1] u2] u2
```

Indeed, the `foo` function takes two ordinary arguments `x` and `y` and a function f which is not ordinary: it is a function which takes two arguments of the same type as `x` and `y` (types `u0` and `u1`). The `foo` function returns a result of the same type as the `f` function.

Here you will note that the use of functions in SCOL is done in a manner completely controlled by typing.

Another way of using functions consists in creating a function from another function and from an argument. We will call this a node. SCOL provides a function that performs this operation:

```
mknode : fun [fun [u0 u1] u2 u1]  fun [u0] u2
```

Example:

```
fun f(x,y)=(atoi x)+y;;
fun g()= let mknode @f 1 -> h in exec h with ["16"];;
```

The `g` function defines an h function which is equal to the `f` function, an argument of which we will have set (here the second).

- the type of f is: `fun [ S I ] I`
- the type of h is: `fun [ S ] I`

To generalize the mknode function, you define functions mkfun1, mkfun2, ..., mkfun8. For example mkfun8 takes a function with 8 arguments and an argument, and returns a function with 7 arguments.

```
mkfun8 : fun [ fun [u0 u1 u2 u3 u4 u5 u6 u7] u8 u7] fun [u0 u1 u2 u3 u4 u5
u6] u8
```

Note that the **mkfun2** function is the same as the **mknode** function.

3.5.9. Redefining functions

In a SCOL file, you can only declare a name once. Furthermore, an `f` function referenced in a `g` function must be defined *upstream* of the `g` function.

When compiling several SCOL files in succession, you can redefine a name defined in a previous file, with the exception of type names.

In some cases, it can be useful to predefine the type of a variable or function. This is essential for variables that include types other than tuples, lists, integers and character strings. It is also essential when two functions call each other.

Example 1:
```
/* definition of an integer list-type variable */
typeof x = [ I r1 ];;
var x= [1 [2 [3 nil]]];;
/* without the typeof, the language would determine the type of x as [I [I
[I u0]]], which contains a free variable */
```

Example 2:
```
/* functions which refer to each other */
proto g= fun[I] I;;

fun f(x)= if x>0 then 1+ g x-1 else 0;;
fun g(x)= if x>0 then 1+ f x-1 else 0;;
```

If you want to initialize a variable to **nil**, it is useless writing: `var x = nil;;`

`typeof` is used both to predefine a type and to initialize the variable to **nil**.

Example 3:
```
/* definition of an empty integer list variable */
typeof x=[I r1];;
fun f()= set x = [1 [2 [3 nil]]];;
```

Conversely, you can define a variable, a prototype, a type constructor or a structure calling on a type which has not yet been defined. You will be able to define this type later (in a subsequent package for example), but only once.

3.5.10. New examples

You are now familiar with all the SCOL language calculation principles. We shall now present a variant of the *Quicksort* program; this time it will be polymorphic, and will be used to delete duplicates (elements present twice in the list).

The idea is fairly simple: you give the `quicksort` function a list to sort and a function for comparing two elements in the list.

This function – which has two arguments (the two elements to be compared) – must return:

- a strictly positive number if the first element is greater than the second
- a strictly negative number if the first element is less than the second

- zero if both elements are equal and provided you want to delete the duplicates from the list. If you do not want to delete the duplicates, simply return any non-null integer when two elements are equal.

The following program performs two sorts: one on a list of integers, another on a list of character strings. The display functions are not polymorphic: there is one to display the list of integers, and another to display the list of character strings.

Here, the type of the `quicksort` function equals:

```
fun [[u0 r1] fun [u0 u0] I] [u0 r1]
```

In the `mystrcmp` comparison function, note the use of the `strcmp` function. This function, which you will find in most programming languages, compares two character strings. It returns 1 if the first is greater (in alphabetical order) than the second, -1 if the first is less than the second, 0 if the two strings are equal.

```
/* MyTest */

fun conc(p,q)=
 if p==nil then q
 else (hd p)::conc (tl p) q;;

fun dividelist (x,p,f)=
 if p==nil then [nil nil]
 else
  let p->[a n] in
  let dividelist x n f ->[r1 r2] in
  let exec f with [a x] -> r in
  if r==0 then [r1 r2]
  else if r<0 then [a::r1 r2]
  else [r1 a::r2];;

fun quicksort(l,f)=
 if l==nil then nil
 else let l->[vl nl] in
 let dividelist vl nl f->[va na] in
 conc quicksort va f vl::quicksort na f;;

/* display list */
fun displayIntList(l)=
 if l==nil then "nil\n"
 else
  let l->[a n]
  in strcat strcat (itoa a) "::" displayIntList n;;

fun displayStrList(l)=
 if l==nil then "nil\n"
 else
  let l->[a n]
  in strcat strcat a "::" displayStrList n;;

fun myintcmp(a,b)=a-b;;
fun mystrcmp(a,b)=strcmp a b;;

fun mymain()=
 strcat
  displayIntList
   quicksort 3::5::2::8::6::5::nil @myintcmp
```

```
  displayStrList
   quicksort "ab"::"abc"::"xy"::"www"::"pqr"::nil @mystrcmp;;
```

The program returns the following result:

```
2::3::5::6::8::nil
ab::abc::pqr::www::xy::nil
```

You will notice that the duplicates have indeed been deleted, i.e., the number 5 which appeared twice in the list to be sorted.

3.5.11. Standard library

In this section, you will find the list of basic functions in the SCOL language. The type is indicated for each function. The functions are grouped into several categories:

- functions on integers
- functions on character strings
- functions on lists
- functions on tables
- functions on floating numbers
- time functions
- console functions

Besides the classic functions found in all programming languages, you will notice in particular the following functions:

- `zip` and `unzip`: compression and decompression of a character string.
- `strextr` and `strbuild`: breakdown of a string into a list of lines, whereby each line is a list of words. This simplifies all problems relating to parsing.
- `_getlongname`: function for hashing (or signing) a character string.

3.5.11.1. Functions on integers

`rand : fun [] I`
Returns a random integer, between 0 and 65535.

`srand : fun [I] I`
Initializes the random series with an integer. Returns 0.

`max : fun [ I I ] I`
Returns the max of two integers.

`min : fun [ I I ] I`
Returns the min of two integers.

`abs : fun [ I ] I`
Returns the absolute value of an integer.

`mod : fun [ I I ] I`
Returns the remainder of the division of one integer by another.

3.5.11.2. Functions on character strings

strlen : fun [S] I
Returns the size of a character string.

For example, `strlen "12abc345de"` returns the integer 10

strcat : fun [S S] S
Concatenates two character strings (the two initial strings are not modified).

For example, `strcat "12" "abc"` returns the string `"12abc"`

STRCATN : FUN [[S R1]] S

Concatenates a list of character strings (equivalent to, but more efficient than, the previous function repeated *n-1* times).

For example, strcatn `"12"::"abc"::"345"::"de"::nil` returns the string `"12abc345de"`

strcmp : fun [S S] I
Compares two character strings (using the standard C function).

strfind : fun [S S I] I
Searches for the first string in the second string from the position passed as a parameter (the first character is in position 0). Returns `nil` if the string was not found, or else the position in which the string was found.

strfindi : fun [S S I] I
The same as above, but without taking into account the differences between upper and lowercase characters.

listtostr : fun [[I r1]] S
Transforms a list of integers into a character string: each link gives a character. The first link gives the first character.

For example `listtostr 65::66:67::nil` returns the string `"ABC"`

strtolist : fun [S] [I r1]
Reverse of the previous function.

For example `strtolist "ABC"` returns the string `65::66::67::nil`

atoi : fun [S] I
Interprets a character string as an integer.

For example, atoi `"53"` returns the integer `53`

itoa : fun [I] S
Reverse of the previous function.

For example, `itoa 53` returns the string `"53"`

ctoa : fun [I] S
Creates a character string containing a single character of ASCII code.

For example, ctoa 65 returns the string `"A"`

HTOI : FUN [S] I

Interprets a character string as an integer coded in hexadecimal (unsigned).

itoh : fun [I] S
Reverse of the previous function.

substr : fun [S I I] S
Returns a substring of the string passed as an argument.
The first integer gives the substring's initial position, whereby the first character is in position 0, and the second integer gives the substring's size.

For example, `substr "abcdef" 2 3` returns the string `"cde"`

strdup : fun [S] S
Creates a copy of a character string in the memory. This is only useful when you modify a character string "on the spot" using the `set_nth_char` function defined below.

strlowercase : fun [S] S
Creates a copy of a character string, replacing the uppercase letters with lowercase ones.

STRUPPERCASE : FUN [S] S

Creates a copy of a character string, replacing the lowercase letters with uppercase ones.

strcmpi : fun [S S] I
Compares two character strings without considering upper/lowercase (using the standard C function).

nth_char : fun [S I] I
Returns the n-th character of a string.

set_nth_char : fun [S I I] S
Modifies the n-th character of a string (danger: the string is modified "on the spot", all the pointers to the string are affected by this modification; this is a side effect).

zip : fun [S] S
Compresses a character string.
The compression rate is 60\% on average.

unzip : fun [S] S
Reverse of the previous function.

strtoweb : fun [S] S
Converts an ordinary character string into one only containing alphanumeric symbols as well as '+' and '%':

- alphanumeric characters are retained
- spaces are replaced by '+'
- other characters are replaced by a '%' followed by 2 hexadecimal digits.

WEBTOSTR : FUN [S] S

Reverse of the previous function

`_getlongname : fun [S1 S2 S3] S`
This function is used to manage files, but can be practical for other uses. It performs a signature.

- S1 is the string to sign
- S2 is an ordinary string
- S3 codes the signature type, using the specific character
- "#": first signature type

This function returns the concatenation of S2, of the signature's specific character and of the signature of the S1 string.

`lineextr : fun [S] [S r1]`
The `lineextr` function breaks down the initial text into lines (the lines are separated by the characters 10 or 13).

`linebuild : fun [[S r1]] S`
The `linebuild` function is the reverse of the previous function. It reconstitutes the text by concatenating the lines, separated by a character 10.

`strextr : fun [S] [[S r1] r1]`
The **strextr** function breaks down the initial text into lines, then each line into words, and builds the result in the form of a list of lines, whereby each line is a list of words. Only lines containing at least one word are retained.

You can insert special characters in words using the \ character:

- \\ for the \ character
- \+' ' (backslash followed by a space) for the space character
- \+decimal number of at least 3 digits for any ASCII code

A \ at the end of the line is used to ignore the new line.

For example, `strextr "abc def\n1 23 456"` returns the double list

("abc"::"def"::nil)::("23"::"456"::nil)::nil

`strbuild : fun [[[S r1] r1]] S`
The `strbuild` function is the reverse of the previous function. It reconstitutes the basic text, replacing the special characters by their equivalent with \

3.5.11.3. Functions on lists

`list builder ' ::'`
List builder operator. For example, to build the list of the first 5 integers:

```
1 :: 2 :: 3 :: 4 :: 5 :: nil
```

`hd : fun [[u0 r1]] u0`
Returns the first element in a list.

`tl : fun [[u0 r1]] [u0 r1]`
Returns the list without the first element.

`sizelist : fun [[u0 r1]] I`
Returns the size of the list.

`nth_list : fun [[u0 r1] I] u0`

Returns the n-th element in a list (the elements are numbered from 0). If the element does not exist, `nil` is returned.

For example `nth_list 3::5::4::6::nil 2` returns the integer 4.

`endlist : fun [[u0 r1] I] [u0 r1]`
Returns the list from the n-th element (the elements are numbered from 0). If the element does not exist, **nil** is returned.

For example `endlist 3::5::4::6::nil 2` returns the list 4::6::`nil`.

3.5.11.4. Functions on tables

`sizetab : fun [tab u0] I`
Returns the size of a given table (the table's number of cells).

`mktab : fun [ I u0 ] tab u0`
Creates a table of a given size initialized with a given value.

`tabtolist : fun [ tab u0 ] [u0 r1]`
Creates a list from a table, taking the elements in increasing index order.

`tabtolistR : fun [ tab u0 ] [u0 r1]`
Creates a list from a table, taking the elements in decreasing index order.

`listtotab : fun [ [u0 r1] ] tab u0`
Creates a table from a list, retaining the order of the elements in the list.

`listtotabR : fun [ [u0 r1] ] tab u0`
Creates a table from a list, inverting the order of the elements in the list.

3.5.11.5. Fonctions sur les nombres flottants

`itof : fun [ I ] F`
Transforms an integer into a floating number.

`ftoi : fun [ F ] I`
Transforms a floating number into an integer (by rounding the floating number).

`ftoa : fun [ F ] S`
Transforms a floating number into a character string.

`atof : fun [ S ] F`
Reverse of the previous function.

`absf : fun [ F ] F`
Calculates the absolute value of a floating number.

`PIf : fun [ ] F`
Returns the constant **pi**.

`cos : fun [ F ] F`
Cosine function.

sin : fun [F] F
Sine function.

tan : fun [F] F
Tangent function.

acos : fun [F] F
Arc cosine function.

asin : fun [F] F
Arc sine function.

atan : fun [F] F
Arc tangent function.

atan2 : fun [F F] F
Arc tangent function with two arguments: "atan2 y x" returns the signed angle formed between the axis of the abscissa and the point (x,y).

Ef : fun [ ] F
Returns the constant **e**.

log : fun [F] F
Logarithm function.

log10 : fun [F] F
Logarithm function in base 10.

exp : fun [F] F
Exponential function.

pow : fun [F F] F
Power function: ``pow x y'' returns x to the power of y.

sqr : fun [F] F
Calculates the square of a floating number.

sqrt : fun [F] F
Calculates the square root of a floating number.

rootn : fun [F F] F
Calculates the n-th square of a floating number: " rootn x y " returns x to the power 1/y.

3.5.11.6. Time functions

time : fun [] I
Returns the number of seconds passed since January 1st 1970.

ctime : fun [I] S
Returns a character string giving the time and date in the following format: ``Tue Jan 21 11:24:53 1997''. The parameter is the number of seconds passed since January 1st 1970, typically issued from the previous function.

_tickcount : fun [] I
Returns the number of milliseconds passed since the machine was started.

3.5.11.7. Console functions

_showconsole : fun [ ] I

Displays the console.

_hideconsole : fun [ ] I

Hides the console.

_fooS : fun [S] S

Sends a character string to the console. Returns the same string. The string must not be more than 4MB.

_fooI : fun [I] I

Sends a hexadecimal integer to the console. Returns the same integer.

_fooIList : fun [[I r1]] [I r1]

Sends a list of hexadecimal integers to the console. Result unchanged.

_fooSList : fun [[S r1]] [S r1]

Sends a list of strings to the console. Result unchanged.

3.6. Global variables of the SCOL machine

The SCOL machine manages a list of resource variables which we will simply refer to as '**resources**'. These variables are defined by their name, and are associated with a character string. They are independent of the channel management, and in particular outlive the channel which defined them.

The API contains two instructions:

_getress : fun [S] S
Finds a resource of a certain name and returns the associated value (**nil** if the resource is not defined).

_setress : fun [S_1 S_2] S
Defines the resource S_1 with the value S_2 and returns S_2. If the resource did not exist, it is created. If it existed, the new value is assigned to it. If S_2 equals **nil**, the resource is destroyed.

SCOL uses a resource initialization file, called **usmress.ini**, located in the SCOL directory (usually C:/Program Files/SCOL). This is a text file containing lines of two words, whereby the first is the name of a resource and the second the corresponding value.

This file is analyzed at the start of the SCOL machine's operation using the following program:

```
fun multiress(res)=
 if res==nil then 0
 else let res ->[[l n] nxt] in
 (if strcmp l "#" then _setress l hd n else nil;
  multiress nxt);;
```

```
...
multiress strextr _loadressini;
...
```

Note the `_loadressini` function:

_LOADRESSINI : FUN [ ] S

Reads the **usmress.ini** file and returns it in a character string. This file is not usually in one of the SCOL partitions, which is why you need a special function to read it.

There are two special variables which can be considered to be resources: the version number and name of the SCOL machine:

`_version : fun [ ] I`
Returns the version number.

`_versionname : fun [ ] S`
Returns the version name.

The order in which the following chapters are presented is of no special importance. It is up to you to establish your own order according to the topics that interest you. If you are interested in the details of the SCOL language, start by reading the sections on Channels and communications and File management.

If you are impatient to start developing your first programs, go straight on to the section dealing with event-driven programming.

4. CHANNELS AND COMMUNICATIONS

In SCOL, the notion of channel is closely linked to the notion of communication. In fact, network connections are systematically linked to an environment to form a channel. In the first section we explain how to create and manipulate channels. In the second section, we show how to use these channels to communicate from one machine to another.

4.1. Manipulating channels

We saw at the beginning of the previous chapter that the SCOL machine is based on the notion of the channel. The SCOL machine is multi-channel. Each channel is a pair (environment, network connection). The network connection is usually a TCP/IP or UDP socket-type connection. However, there may be no network connection, in which case it is called **unplugged**.

Channels and environments are objects that can be easily manipulated in SCOL. Two types are therefore associated with them:

- the `Chn` type represents a channel.
- the `Env` type represents an environment.

4.1.1. Channel manipulation API

`_channel : fun [] Chn`
Returns the current channel.

`_script : fun [S] I`
Runs a script in the environment of the current channel. Each line of the script is in the form of command-arguments: the range is dynamically defined. The result is 0. An unknown command or a mistake in typing the parameters are not considered as errors: the command is simply skipped. The exact syntax of a script is defined further on, but we can already say that it as the same as the **.SCOL* files (see the 'Hello World' examples in chapter II).

`_scriptc : fun [Chn S] I`
Same as above, but in a different channel, which is specified by parameters.
Note: `_script X` is equivalent to `_scriptc _channel X`

`_load : fun [S] I`
Loads and compiles a SCOL file (also called package) in the current channel whose name is passed as a parameter (see further on under file management).

`_setchannel : fun [Chn] Chn`
Changes the current channel (to be used with caution).

4.1.2. Environment management API

`_envchannel : fun [ Chn ] Env`
Returns the environment associated with a channel.

`_removepkg : fun [ Env ] Env`

Returns the environment from which you retrieved the package located at the start (i.e., the last package compiled).

_envfirstname : fun [Env] S
Returns the name of the first package in this environment. By using this function several times with `_removepkg`, you can find out the names of every package in an environment.

_setenv : fun [Chn Env] I
Changes the environment of a channel. If the environment is **nil**, the channel will retake the **minimal environment**.

4.1.3. Creating and destroying a channel

4.1.3.1. TCP/IP channel

Since a channel is a pair (environment, network connection), to create a channel you have to specify both elements of the pair. To do this you use the `_openchannel` function. This function creates either an unplugged channel or a channel with a TCP/IP connection.

This function uses three arguments:

- the connection address (`nil` if the channel is **unplugged**). The address is a string comprising the IP address of the server to be contacted and the port number, for example: "123.234.54.34:1025". If the **IP address** is omitted, the connection is made locally (":1025" is equivalent to"127.0.0.1:1025")
- a script (the syntax is defined further on)
- the environment which the channel will inherit. If the environment is `nil`, the channel inherits the **minimal environment**.

The function returns the channel created. This is `nil` in the event of an error.

_openchannel : fun [S S Env] Chn

The `_openchannel` function operates as follows: a new channel is created with the environment passed as a parameter and the possible network connection. Then the script is run on this new channel.

Example: you want to create an unplugged channel which inherits the environment of the current channel, and in which you want to compile the package "new.pkg" and run the main function located in this package. You need to write:

```
_openchannel nil "_load \"new.pkg\"\nmain" _envchannel _channel
```

The following version is equivalent:

```
_scriptc
 (_openchannel nil "" _envchannel _channel)
 "_load \"new.pkg\"\nmain"
```

In order to open a channel to port 2000 of a machine whose IP address is 1.2.3.4, give it the minimal environment, compile on it the package *"new.pkg"* and run the main function located in this package, you need to write:

```
_openchannel "1.2.3.4:2000" "_load \"new.pkg\"\nmain" nil
```

4.1.3.2. UDP channel

You can create a channel whose network connection is a UDP listening socket:
`_setUDP : fun [ Env I S ] Chn`

This function takes an environment, a port number and a script. A UDP channel is created on the port whose number is given, with the specified environment in which you begin by running the script.

Remember that UDP messages are not totally reliable: transmission and order of arrival are not guaranteed. However, because there is no buffering, the latency of transmission is negligible.

The UDP channel is not connected to another channel: several correspondents can send messages to this channel, and there is even a broadcast potential.

4.1.3.3. Destroying channels

`_closechannel : fun [] I`
Closes the current channel.

`_killchannel : fun [Chn] I`
Closes the specified channel.

`_closemachine : fun [] I`
Closes all channels, and therefore stops the SCOL machine.

4.1.4. Creating and destroying a server

We have just seen how to open TCP/IP channels using the `_openchannel` function. To do this, we saw that you have to specify the correspondent's IP address and port number. This means that the correspondent "waits for" this type of connection. In network terminology, the correspondent is said to have opened a **server** on this port. Obviously, the SCOL language can be used to open and manipulate these servers. These are `Srv`-type objects.

You use the following function to define a server:

`_setserver : fun [Env I S] Srv`

Opens a server on the port number and with the script passed as parameters. Returns **nil** if impossible (the port is probably already being used by another server). The server thus created inherits the environment passed as a parameter.

When a connection to this server is requested, i.e., when another machine has run the `_openchannel` function with this server's address, a new channel is created. This channel inherits the environment defined with the server. The script defined with the server is then immediately run in this new channel.

Once created, a server channel is indistinguishable from a client channel. This is one of SCOL's key concepts: once the connection is made, communication is symmetric.

For example, let us suppose that the following function is called on Alice's machine:
`_setserver _envchannel _channel 2000 "_load \"new.pkg\"\nmain"`

A server is created on port 2000. The environment defined with the server is the current environment. The script defined with the server compiles the *"new.pkg"* file and runs the `main` function.

When a connection is opened from another machine, let's say Bob's, a channel is created on Alice's machine, which inherits the environment and in which the script is run. There is now on Alice's machine a channel that is connected to a channel on Bob's machine. This connection is symmetric and allows data to be exchanged between both machines.

You use the following function to stop a server:
`_closeserver : fun [Srv] I`

Destroys the server passed as a parameter.

NB: this only ends the potential to receive new calls on this server and does not destroy the channels that have been created from it.

4.1.5. Additional functions of channel management

`_channelname : fun [Chn] S`
Returns the complete address of a correspondent of a channel (IP address and port number): for example, "127.0.0.1:1234".

`_channelIP : fun [Chn] S`
Returns the correspondent's IP address only: in the previous example, "127.0.0.1" only.

`_channelport : fun [Chn] I`
Returns the correspondent's port number: in the previous example, 1234.

`_gethostbyname : fun [S] S`
Common function that carries out address resolution in the direction *machine name->IP address*. NB: this is a **blocking** function: it may stop the machine for several seconds if resolution is difficult or impossible.

`_getnamebyIP : fun [S] S`
Common function that carries out address resolution in the direction *IP address->machine name*. NB: this is a **blocking** function: it may stop the machine for several seconds if resolution is difficult or impossible.

`_hostname : fun [] S`
Returns the host name. If the host has no defined name, returns ``localhost''.

`_hostIP : fun [] S`
Returns the host's IP address: if the host does not have one, returns "127.0.0.1".

`_channeltime : fun [ Chn ] I`
Returns the time elapsed since the channel was created (in seconds).

`_servertime : fun [ Srv ] I`
Returns the time elapsed since the server was created (in seconds).

4.1.6. Script syntax

As we indicated earlier, here we specify script syntax. Scripts are used in a number of cases:

- start-up **.SCOL* files on the SCOL machine.
- `_script` functions and `_scriptc`.
- the functions which create channels and servers, such as environment initialization scripts.

A script is a character string ending in 0.

A script is made up of one or more lines: the lines are separated by the characters 10.

Each line comprises words separated by code characters less than or equal to 32.

The first word is a mnemonic, which normally corresponds to a function name.

The following words are:

- either `NIL`
- or a hexadecimal integer (8 figures at most)
- or a character string beginning and ending with a `"`

In the character string, the backslash character (\) has a particular significance:

- `\n`: 10 character
- `\z`: 0 character
- `\ decimal number (three figures at most):` any code character
- `\other`: the backslash is ignored and you skip to the next code character greater than or equal to 32

Thus for the \ character, we shall use `\\`. For the " character, we shall use `\"`.

Each line of script is interpreted as a command followed by arguments. These arguments may be either integers, character strings, or NIL (the only possibility for the other types).

NB:

**.SCOL* files must not be too long: in fact, their size is limited by the size of the command line accepted by *Windows*. Restrict yourself to three or four lines. In other uses of scripts, there is no limit to the size of the script.

Example for a **.SCOL* file script:

```
_load "foo.pkg"
main 1234 "qsd\\q\"sdf\n\0\12\234\0122" NIL
```

Same script used with the `_script` function:

```
_script "_load\"foo.pkg\"\nmain1234
\"qsd\\\\q\\\"sdf\\n\\0\\12\\234\\0122\" NIL"
```

4.2. Communications in SCOL

4.2.1. Controlling connections: particular events

Opening a channel with the `_openchannel` function takes a certain amount of time (up to a few seconds). The computer has to find its way through the network, then communicate with the other machine to check that a server is indeed open. This raises the question: when is the connection really established?

The `_openchannel` function is in fact virtually instantaneous: the channel is created immediately, the network connection is started and the program continues before the connection becomes fully active.

When the connection is made, the SCOL machine looks for a '`_connected`' name function of the type:

```
fun [ ] ?
```

(The question mark means that the machine is not concerned with the result type of the _connected function.)

If the _connected function has not been defined, nothing happens and the machine continues as normal, otherwise the _connected function is run.

The same thing happens on the server: a channel is created on the machine that hosts the server, the script is run on it, and the possible _connected function is then triggered on it.

Another network event may occur unexpectedly: disconnection. This may be due either to the correspondent destroying the channel or shutting down the machine, or it may be due to a network problem. In this case the SCOL machine closes the channel which lost its correspondent, but before it does so it looks for a '_closed' name function of the type:

```
fun [ ] ?
```

If the _closed function has not been defined, nothing happens and the machine continues as normal, otherwise the _closed function is run.

Similarly, if the server has reached the maximum number of simultaneous connections that it can handle, the _fullserver function is run in the server's environment.

4.2.2. Sending a message using the _on function

Once a channel is created, messages may be sent in both directions. The messages that circulate are always in the form of "command arguments". In fact, they take the form of a single-line script. When a message arrives, the SCOL machine looks in the environment of the channel for a function whose name is "__command" (the double underscore is added by the receiving machine and ensures that only functions beginning with a double underscore can be activated by the correspondent). If this function exists, the SCOL machine checks the type of arguments. If everything corresponds, the function is run and the result is lost: only the side effects will count.

The SCOL language provides a simple solution for using the channel in the send direction. Let's take the following example: Bob wants to send on the channel AliceChannel the foo command with three parameters: two integers, 123 and 345, and a string, "bar".

The function you use is called "_on", it is of the type: fun[Chn Comm] I.

Note: messages sent on an **unplugged** channel are simply ignored.

With the _on function, the preceding example is written:

```
defcom X = foo I I S;;
...
  _on AliceChannel X [123  234 "bar"];
...
```

Subsequent to **defcom**, the compiler defines a **communication constructor** X of the type:

```
fun[[I I S]] Comm
```

X is thus a function that takes a tuple of three elements (an integer, an integer, then a string) and returns a message containing these three arguments preceded by the "foo" command. Communication builders are used only to pass integers (I) and character strings (S).

Let us define the _on function:

```
_on :  fun[Chn Comm] I .
```

To make it simpler to write, the same name may sometimes be given to the command and to the communication constructor:

```
defcom foo = foo I I S;;
...
  _on AliceChannel foo [123  234 "bar"];
...
```

NB: messages produced by communication constructors for sending using the `_on` function should not be too long (less than 8 Kb).

There is a variant with **`defcomvar`**. This variant is particularly good for the use of "callbacks". For example, let there be a "`call`" function which takes a string and an integer at the start and has to send a message with an argument that is the integer and a command that is the string. The static definition of the command name with `defcom` cannot solve this case. The `defcomvar` function must then be used:

```
defcomvar Y = I;;
fun call(s,i)=
  _on AliceChannel Y s [i];;
```

The `Y` function created here has the type "`fun[S  [I]]  Comm`". The string that it waits for will be used as a command.

The example in the previous paragraph is thus written:

```
defcomvar Y = I I S;;
...
  _on AliceChannel Y "foo" [123  234 "bar"];
...
```

4.2.3. Controlling message queues

When a message is sent by the **`_on`** function on a TCP/IP-type channel, it is placed in a queue particular to the channel. The SCOL machine tries to send it as quickly as possible. The size of the queue (called fifo: First In First Out) can usefully be controlled by means of two functions:

`_waitingfifo : fun [ Chn ] I`
Returns the number of messages present in the queue.

`_sizewaitingfifo : fun [ Chn ] I`
Returns the size in bytes of all the messages present in the queue.

`_setsizefifo : fun [ Chn I ] Chn`
Defines a maximum size for fifos (nil: no size limit). If the size of the fifo is exceeded during an _on, the channel is disconnected and a _closed event will occur.

Note: the SCOL fifo is located between the SCOL application and the computer's network layer fifo, whose size is difficult to calculate. Restricting the size of fifos is primarily to protect the SCOL machine from having insufficient memory, since channel fifos and SCOL applications coexist in the same memory strip.

4.2.4. Sending a UDP message

We have seen how to create a UDP channel with the _setUdp function. This channel is actually a UDP server: it listens on a particular port for UDP messages that are sent to it. These messages are also Comm-type objects, and when a message arrives, the SCOL machine looks in the environment of the channel for a function whose name is "__command", exactly as with the _on function.

To send a message to a UDP channel, you use:

_sendUDP : fun [S Comm] I

You can also send a UDP message via a UDP channel (created by **_setUDP**), using the following function. This may be useful for getting past some proxies in the outer to inner direction: in fact, some links remember that an internal UDP "server" sent a message to an external UDP server, and then authorize it to respond, which enables UDP messages to get past the link in the outer to inner direction:

_sendUDPchn : fun [Chn S Comm] I

The string contains the correspondent's address (in the same format as for _openchannel). A broadcast address may be used.

In contrast to a TCP/IP channel, a UDP channel is not associated with a single machine: everyone can send it a message. It may be useful on the UDP channel to know the IP address of the person who sent it a message. For this you use the _channelIP function described above. In fact, the value returned by this function is updated every time a message is received.

4.2.5. Another use for communication constructors

Above we mentioned the similarity of syntax between Communication (Comm) messages and scripts, specifying that a communication is in fact a single-line script. Moreover, we saw that on the one hand communication constructors make it particularly easy to define a Comm message, while on the other hand the syntax of scripts is rather complicated.

A program might need to build a character string itself that will be used as a script. It would not be easy to build it "manually" using functions on character strings, especially if the script contains character-string arguments that include special characters such as " and \, or if the script contains integer arguments to be written in hexadecimal notation.

To simply this task, the SCOL language provides a link between Comm objects and S character strings; the following function is used to convert a Comm structure into a string that can be used by a script:

mkscript : fun [Comm] S

The string obtained contains a line of script followed by a carriage return, which means that lines of script can be concatenated easily.

Example :

```
defcom loadScr = _load S ;;
defcom mainScr= main S I ;;
...
_openchannel  ":1234"
      strcat (mkscript loadScr ["foo.pkg"] )
            (mkscript mainScr ["bar" 123] )
      _envchannel _channel ;
...
```

The parentheses are unnecessary here.

On running, the second argument of the `_openchannel` function will be:

```
“_load \”foo.pkg\”\nmain \”bar\” 7B”
```

4.3. Methodology of network programming in SCOL

The principle of communication in SCOL is thus as follows: machines exchange messages with each other on channels. When a machine receives a message on a channel, it looks to see if this message is of a sort that will trigger a process: for this, a function is needed that corresponds to the message, with the right number and the right type of arguments.

One of the axioms of SCOL technology is that it is impossible to be sure of the correspondent’s integrity. In concrete terms, what this means is that if A sends B a message X supposedly to receive a response Y, there is no guarantee that response Y will in fact be sent. The only thing A is certain of is how it processes the messages it receives. In this sense, communication in SCOL assumes a human character: when you speak, you can never be sure that you are understood, regardless of the effort you make. It may be objected that often machines A and B are programmed by the same person and that, in this sense, the developer knows very well that B will respond with message Y when it receives message X. However, even in this case, the precautionary principle should apply since you will thereby guard against:

- your own errors. In the previous example, if you programmed A **and** B yourself so that A sends message X and B responds with message Y, it is quite possible that you made an error and that B does not respond. This is all the more likely given that communications, being side effects, cannot be verified statically.
- acts of malice. One or other of the correspondents A or B may have been modified with an intent to cause damage. In this case, A may receive a message Y in "response" to a message X that it has not sent. Similarly, B may receive a message X at a time other than that anticipated by the developer.

To create a network application, it is a good idea to apply the following methodology:

1. Do a communications diagram, i.e. write a list of the messages exchanged between correspondents. Specify the function, number and type of arguments. Your diagram should clearly show the dialog between machines, rather like the script for a play.

2. Specify how the machine should behave when it receives a message. Your specification should not presuppose the time at which the message is received: it should cover every eventuality, even if this seems unnecessary, for the simple reason that a message may arrive at any time, either because of a malfunction or through a malicious act.

Quite often, this behavior will specify that one or more messages be sent which might wrongly be considered as a response.

3. Write down a list of `defcom`

4. Write down the functions in SCOL which code this behavior. The situation is reversed in some way by appearing to consider the SCOL program as a side effect of the message received.

In the log file, you can review all the messages received, preceded by the word `"exec"`. When an error occurs, the file indicates that the message cannot be interpreted either because it does not correspond to any function, or because the number or the type of arguments is incorrect. In this case, the message is simply ignored.

5. FILE MANAGEMENT

The SCOL machine has limited access to the files on your computer. Access is established by defining one or more directories known as **SCOL partitions**. The SCOL machine can only access files located in one of these directories or in one of their subdirectories. At a given moment, only one of the SCOL partitions will be accessible in write mode.

5.1. COL partitions

A SCOL partition is simply a directory of the disk to which the SCOL machine has access. It may also access all of its subdirectories.

Several partitions may be defined. You are recommended to define at least 2, as the first one has a special role. These are usually as follows:

- The first partition is the cache partition: when a user visits a site, this is the partition in which the files downloaded from this site will be stored. A quota may be defined for this partition: the system will check that the number of files present in this partition does not exceed the quota.
- The second partition is your working partition: this is the partition in which your tools and your own creations are located. Your tools may write in this partition.
- The subsequent partitions are additional caches.

SCOL partitions operate along the following principles:

- When the machine searches for a file, it searches in the first partition, then in the second, then in the third. The search stops when the file is found. A file can thus hide another file with the same name and located in a later partition.
- When the machine writes a file, it writes it in the first partition.
- When you start the machine, the first partition is ignored: the cache partition is deactivated. This means that the file search starts at the second partition and that a file is systematically written in the second partition.
- You can activate the cache partition at any time: from this moment on, the file search starts at the first partition, and a file is systematically written in the first partition.
- You cannot deactivate the cache partition once it has been activated.

Thus the underlying security mechanism is clear: each time a user connects to a site, the cache partition will be activated: all files will be written in this partition. As this operation cannot be reversed, the site cannot deactivate the cache partition and write in the other partitions.

Partitions on the SCOL machine are also defined in the **usm.ini** file located in the SCOL directory (usually *C:/Program Files/SCOL*). This is a text file in which the lines beginning with 'disk' each define a partition: a path, which may be followed by a decimal number indicating the quota. As we have just seen, the order is important: the first partition defined will be the cache partition.

```
# sample extract from usm.ini file

disk ./Cache 32768
disk ./Partition 0
disk c:/cdrom
```

The figure 0 opposite the `/partition/` partition indicates that the working partition is accessible in write mode.

The function used to activate the cache is:

```
_cacheActivate : fun [ ] I
```

You can modify the path of a partition dynamically. This can only be done in a highly specific way, by **extending** the partition path. For example, if the partition was *C:\SCOL\A*, you can change it to *C:\SCOL\A\1*, but you cannot obtain *C:\SCOL\B*. You use the _refine function, which just extends the path of the first partition (possibly by adding the missing '/'). This suffix must not contain some character sequences `('..',' ~','//', ...)`.

```
_refine : fun [S] S
```

This function is useful for managing a multi-user system. Several SCOL machines can thus operate simultaneously in different partitions by being connected to the same SCOL Voy@ger (see the SCOL Voy@ger section further on). The SCOL Voy@ger starts the users' machines simply by inserting at the beginning of the start-up script for these machines a _refine function with the subdirectory dedicated to the user as an argument.

5.2. File types

SCOL provides two file types: **normal** files and **signed** files.

5.2.1. Normal files

In this mode, the user gives the file name without restriction. In read mode the file is searched in the different partitions, while in write mode the file is placed in the partition which is accessible in write mode (cache partition or working partition). This mode provides a low level of security: a file is accessible in read and write modes as soon as its name is known. Characters that may be used to write a file name are: alphanumeric characters, period, underscore, space, tilde, dash and slash. For security reasons, the system has the following restrictions:

- two consecutive periods are not allowed
- two consecutive slashes are not allowed
- the name should not begin with a slash
- the name should not begin with a tilde
- a tilde should not follow a slash

5.2.2. Signed files

5.2.2.1. Hash function signature

In this mode, the user gives a clear text file name, as defined above, to which the system appends a cryptographic-type signature on the file's contents. This signature starts with a special character that determines the type of signature (in the first signature implemented, the '#' character), then continues and ends with a succession of alphanumeric characters.

It is therefore virtually impossible to guess the name of any file if its contents are unknown. Furthermore, the contents of a file of this type cannot be modified: any modification results in a change of signature, and hence a change of name. Signed files thus provide a high level of security.

Another useful feature of the signature is that it quickly and accurately determines whether a file has already been loaded; usually, when users contact a SCOL server, they are given a list of the packages they will need. By using the signature, users can find the exact packages they already have in their partitions. In particular, this automatically solves the problem of updating.

5.2.2.2. Cookie signature

You can sign a file with any word: it will be separated from the clear text file name by the ';' character. This word will be called the 'cookie' word. A SCOL machine can define a 'cookie' word once by using the following function:

`_setCookies : fun [S] S`

Given that you cannot redefine a 'cookie' word, this function is used to protect certain files. Usually, a machine used in standard client mode will use the server's IP address as the cookie word, even before it compiles the packets indicated by the server. Files signed with this cookie will be inaccessible to clients of other servers.

For security reasons, when a SCOL machine which has activated its cache starts another SCOL machine, this new machine:

- automatically activates its cache itself on start-up
- defines the cookie '__'

5.3. File management API

The **P** type has been created to use files (P is for *Path*). It remains globally invisible to the user. For special operations in write mode, the **W** type is introduced.

To read a file you need to know its name relative to the SCOL partitions (S-type character string). The `_checkpack` function is used to search for this file in the different partitions and to return a P-type object which actually contains the exact path of the file. You can then use this P-type object for read operations. The only exception is the `_load` function, which is used to compile a package; it performs both operations itself (looks for the file and reads).

To write a file, you need to know its name (S-type character string). If you already have the contents of the file to be written (in a character string), you use the `_storepack` function. If you want to write the file in several stages (streaming), you use the `_getmodifypack` function to obtain a W-type object which will be used by the `_createpack` and `_appendpack` functions.

In SCOL there is no means of knowing the complete path of a file; indeed, this would pose a security problem.

`_checkpack : fun [S] P`
Searches for a package in the partitions from the complete name (file name in clear text followed possibly by a signature). Returns `nil` if not found.

`_getpack : fun [P] S`
Loads a file in a character string. To obtain the file path, you first need to have calculated a `_checkpack`. If the path is `nil`, the result is also `nil`.

_PTOSCOL : FUN [P] S

Returns the complete SCOL name (relative to the partition) of a file: this operation is the opposite of `_checkpack`. Returns `nil` if the file is not in one of the partitions.

`_GetFileNameFromP : fun [P] S`
Returns the file name only, without the path.

`_GetFileNameFromW : fun [W] S`
Returns the file name only, without the path.

`_storepack : fun [S S] I`
Saves the first argument with the second as a name. When this name has a signature, the signature is checked. The result is 0 if successful. The subdirectories contained in the name are automatically created if necessary.

`_getmodifypack : fun [S] W`
Resembles the `_checkpack` function. However, it prepares for write mode with the following two functions. In particular, the file to be created or modified cannot be signed by its contents.

`_createpack : fun [S W]  I`
Opens the W file in write mode, by reinitializing it and writing the S string.
Returns 0 if successful, -1 in the event of an error.

`_appendpack : fun [S W]  I`
Opens the W file in write mode and adds the S string at the end of the file.
Returns 0 if successful, -1 in the event of an error.

`_WtoP : fun [W] P`
Converts W type to P type. The opposite is not possible, for security reasons.

`_load : fun [S] I`
This function has already been described. It loads and compiles a file containing the SCOL code. In fact, the file is searched in the SCOL partitions of the machine (the `_load` function calls the `_checkpack` function itself).

`_getlongname : fun [S1 S2 S3] S`
Calculates the complete name of a file:

- S1 is the file to be saved
- S2 is the file name in clear text
- S3 codes the type of signature by taking up the specific character
- "": no signature
- "#": first type of signature by contents
- ";": cookie-type signature

This function returns the complete name (name in clear text possibly followed by the signature).

This function has already been described in the standard library relating to character strings.

```
/* Example of handling files */
/* reads a file foo.pkg and recopies it under the name 'bar#signature',
   checking the result at each stage */

fun main()=
 let _checkpack "foo.pkg" -> path
 in if path==nil then -1
 else let _getpack path -> n
```

```
in if n==nil then -1
else let _getlongname n "bar" "#" -> n2
in if n2==nil then -1
else _storepack n n2;;
```

`_listoffiles : fun [S] [S r1]`
Returns the list of files present in a directory. The files are returned with their complete name. This function is only accessible if the machine has not yet activated its cache.

`_listofsubdir : fun [S] [S r1]`
Returns the list of subdirectories in a directory. Subdirectories are returned with their complete name. This function is only accessible if the machine has not yet activated its cache.

5.4. Advanced file-reading functions

It is useful to have more refined functions for reading files. For this purpose, a **`File`** type is defined which corresponds to an open file in read mode. This object is obtained using the `_FILEOpen` function from a **`P`**-type object. This object is also associated with a channel: it will be automatically destroyed when the channel is destroyed.

`File _FILEOpen (channel Chn, file P)`
Opens a file in read mode. Returns `nil` in the event of an error.

`I _FILEClose (file File)`
Closes a file.

`I _FILESeek (file File, position I, mode I)`
Positions the start of the read.

`I _FILETell (file File)`
Returns the start of the read position.

`I _FILESize (file File)`
Returns the file size.

`S _FILERead (file File, size I)`
Reads a number of characters in the file, from the current position.

5.5. File selection interface

You can call the file selection graphic interface in read mode or in write mode. Refer to the reference manual for details.

6. EVENT-DRIVEN AND GRAPHIC INTERFACE PROGRAMMING

Here we shall describe the basic principles of event-driven and graphic programming:

- event-driven programming: how to manage the events that the SCOL machine receives (timers, human-machine interface, etc.)
- graphic programming: creating graphic and, more generally, multimedia objects

We shall use two examples to illustrate these principles: windows and (a non-graphic example) timers.

The SCOL graphic interface is the machine's largest API. It is used to manage windows, text zones, buttons, lists, menus, character fonts, bitmaps, etc. It is described in detail in the reference manual.

6.1. Basic principles

6.1.1. Proprietary channel

We refer to an **object** to designate any resource (graphic or otherwise) "external" to the SCOL language: window, button, bitmap, timer, etc.

In SCOL, a channel, said to be proprietary, corresponds to each object. Thus, when you create an object (window, button, timer, etc.), you must specify the object's **proprietary channel**. The object's existence is linked to the channel's existence. The existence of the object will be associated with that of the channel. When you close a channel, all the associated objects are quite simply destroyed.

In the File management section we saw the example of `File`-type files. To create such an object, we used the `_FILEOpen` function, type `fun [Chn P] File`. Here you can see how the proprietary channel has been specified: it is the first argument.

Similarly, the type of the _CRwindow function encountered in the 'Hello World' examples is:

```
_CRwindow : fun [Chn ObjWin I I I I I S] ObjWin
```

The `ObjWin` type corresponds to a window object. To create a window, you thus specify the proprietary channel, followed by the parent window, then by various arguments that define the window's position, size, type and title.

The role of the proprietary channel goes beyond merely destroying objects when the channel is closed. It also involves the management of events.

6.1.2. Managing events

Most objects are likely to receive events. For example, you can click on a window. You can move a window. You can press a button. Each of these actions corresponds to an event which the program must process. Similarly, the role of a timer is to initiate an event at regular intervals.

6.1.2.1. Different event management systems

The following question arises: how does a SCOL program receive the events?

This question is posed for any operating system; however, the answers are often different.

In Windows, the program defines a function which is called each time an event is produced by the system, whatever the event. In UNIX, with X-Window (X11), the principle is similar, but users can choose which events their function is to receive. In both cases, the function "sorts" the events and, according to the type of event, processes it accordingly. In C, this function can often be summed up by a giant 'switch'.

With Xt Intrinsics (high-level *X-Window* programming model, used for example for Osf-Motif), the developer defines one function per type of event. Such a function will only be called when a given type of event occurs. Such a function is called a '**reflex**' or a '**callback**'. This method offers two advantages: on the one hand, you no longer have to write the 'switch' function; on the other hand, the event's parameters are passed more simply. For example, three parameters characterize a click event: the coordinates of the click and the button number. No parameters characterize a timer event. One parameter characterizes an event such as 'the content of a text field has changed', i.e., the new text. Processing all these events by the same function poses a problem in terms of passing parameters: you often (Windows, *X-Windows*) have to take liberties with the typing, which is a major cause for error. The advantage of the system of reflexes is to be able to define one reflex function type per event type. As you will no doubt have guessed, this is the way in which events are managed with SCOL.

6.1.2.2. Defining reflex functions

In SCOL, you can therefore define a reflex function for each object and each type of event. This reflex function must take at least two arguments:

- the first is the object affected
- the second is a given user parameter

Depending on the events, the reflex function will have additional arguments: 3 integers for a click, two integers for a 'resize window' event, no additional arguments for a timer event.

To define a reflex, function, you will use the function with the appropriate definition. For example, in the 'hello3' program described in the 'Hello World' section, you will find the following line:

```
_CBwinDestroy win @_end nil;
```

The `_CBwinDestroy` function is used to define the reflex associated with the 'destroy window' event. It takes three arguments:

- the window whose reflex you are defining (here '`win`')
- the reflex function (here '`@_end`')
- -the user parameter (here '`nil`')

The type of the reflex function is: `fun [ObjWin?]?`, whereby the second argument is the user parameter. The type of the result is irrelevant. The type of the `_CBwinDestroy` function is therefore:

_CBwinDestroy : fun [ObjWin fun [ObjWin u0] u1 u0] ObjWin

You will notice how the typing ensures that the user parameter supplied when the reflex is defined has the same type as that of the reflex function (here '`u0`'). Here, SCOL holds an advantage over Xt-Intrinsics: the user parameter in Xt-Intrinsics is not typed: it forces the developer to play around with types, which is always a major cause of errors.

Other examples:

A click event on a window requires three parameters: the coordinates of the click and the number of the button. The associated reflex function is of the type: `fun [ ObjWin u0 I I I]?`

The definition function of the click reflex function is:

```
_CBwinClick : fun [ObjWin fun [ObjWin u0 I I I] u1 u0] ObjWin
```

A timer is a `Timer`-type object.

The reflex function associated with the timer is of the type: `fun [ Timer u0]?`

The definition function of the timer reflex function is:

```
_rfltimer : fun [Timer fun [Timer u0] u1 u0] Timer
```

If you define a reflex with `nil` as the second argument (instead of the reflex function), this deletes the reflex. Similarly, you can redefine a reflex at any time, including in the reflex function itself.

6.1.2.3. Processing an event

When an event occurs, the SCOL machine searches for the object concerned by the event, then the reflex function associated with this event. If this reflex function has been defined, the SCOL machine runs this function and forgets the result, except in certain highly specific cases where the result is interpreted by the system. This function must be run in a channel (indeed, this function may call the `_load` or `_closechannel` functions). It is the **proprietary channel** that is selected. This explains the second role of the proprietary channel.

You can change an object's proprietary channel; more exactly, you can assign all the objects of a channel to another channel using the following function:

_chgchn : fun [Chn$_1$ Chn$_2$] Chn$_2$

Assigns all the objects of the Chn$_1$ channel to the Chn$_2$ channel.

This function is used in particular to save a channel's objects just before the channel is destroyed (subsequent to the `_closed` event, for example).

6.2. Examples

6.2.1. Windows

Here, we will illustrate in windows the principles that have just been expounded. You will find the complete documentation for the following functions in the 'Graphic Interface' appendix.

```
ObjWin _Crwindow( Chn channel, ObjWin parent, I posx, I posy,
 I taillew, I tailleh, I flags, S name)
```

This function creates a new window.

```
I _DSwindow ( ObjWin fenetre )
```

This function destroys a window.

```
ObjWin _CBwinPaint ( Objwin fenetre, fun [ ObjWin u0 ] u1funreflex, u0 param )
```

Reflex of the "*Paint"* event. No special argument.

```
ObjWin _CBwinMove ( ObjWin fenetre, fun [ ObjWin u0 I I ] u1funreflex, u0 param)
```

Reflex of the "*Move"* event. Two arguments which give the new position.

```
ObjWin _CBwinSize ( ObjWin fenetre,   fun [ ObjWin u0 I I ] u1 funreflex, u0 param)
```

Reflex of the "*Size"* event. Two arguments which give the new size.

```
ObjWin _CBwinClick ( ObjWin fenetre, fun [ ObjWin u0 I I I] u1 funreflex, u0 param)
```

Reflex of the "*Clic"k* event. Three arguments which give the coordinates of the click and the mouse button used.

```
ObjWin _CBwinUnclick ( Objwin fenetre, fun [ ObjWin u0 I I ] u1, funreflex, u0 param)
```
Reflex of the "*UnClick"* event. Three arguments which give the coordinates of the unclick and the mouse button used.

```
ObjWin _CBcursorMove ( ObjWin fenetre, fun [ ObjWin u0 I I I] u1 funreflex, u0 param)
```
Reflex of the "*CursorMove"* event. Three arguments which give the coordinates of the mouse and the mouse button used.

```
ObjWin _CBwinKeydown ( ObjWin fenetre , fun [ ObjWin u0 I I ] u1 funreflex, u0 param)
```
Reflex of the "*KeyDown"* event. Two arguments which give the 'scancode' of the key pressed and the value of the key pressed. See the appendices for this argument's special values.

```
ObjWin _CBwinKeyup ( ObjWin fenetre, fun [ ObjWin u0 I ] u1funreflex, u0 param)
```
Reflex of the "*KeyDown"* event. An argument which gives the 'scancode' of the key released.

```
ObjWin _CBwinDestroy ( ObjWin fenetre, fun [ ObjWin u0 ] u1funreflex, u0 param)
```
Reflex of the "*Destroy"* event. No special argument.

```
ObjWin _CBwinFocus ( ObjWin fenetre , fun [ ObjWin u0 ] u1 funreflex, u0 param )
```
Reflex of the "*Focus"* event. No special argument.

```
ObjWin _CBwinKillFocus ( ObjWin fenetre , fun [ ObjWin u0 ] u1 funreflex, u0 param )
```
Reflex of the "*Kill Focus"* event. No special argument.

```
ObjWin _CBwinDClick (ObjWin fenetre,fun [ObjWin u0 I I I] u1 , u0 param )
```
Reflex of the "*DoubleClic"k* event. Three arguments which give the coordinates of the click and the mouse button used.

```
ObjWin _CBwinDropFile ( ObjWin fenetre, fun [ObjWin u0 I I [P r1]]u1 u0)
```
Reflex of the "*DropFile"* event. Three arguments which give the coordinates of the click and the list of files "deposited" in the window by the user.

6.2.2. Timers

You can define timers, whose period is defined in milliseconds. A timer with a period of 1000 will initiate a timer event every second. Timer objects are of '`Timer`' type. Three functions are used to manage the timers:

`_starttimer : fun [Chn I] Timer`
Creates a timer by defining the proprietary channel with a certain period expressed in milliseconds. Returns the timer object '`nil`' in the event of an error.

`_deltimer : fun [Timer] I`
Destroys the timer passed as an argument.

`_rfltimer : fun [Timer fun [Timer u0] u1 u0] Timer`
Defines the reflex function associated with the timer.

7. 3D PROGRAMMING

The SCOL machine contains a library that is capable of processing and displaying 3-dimensional scenes. This library is called a '**3D Voy@ger**'. Very simply, it is used to add 3D functionalities to your programs. One of the key features of the SCOL language is its skillful combination of Internet communication capacities with powerful 3D functionalities.

SCOL's 3D Voy@ger was carefully developed for on-line use: the scene description files are not large and scenes are rendered fluidly on less powerful machines. Indeed, it is worth pointing out that many Internet users, unlike the users of video-games, don't have up-to-the-minute machines. Similarly, it is impossible to force users on the Internet to acquire a 3D card, or even to be sure that they have the latest driver for their card.

To be able to understand this chapter, you should already have some familiarity with the concepts linked to 3D (scenes, polygons, materials, textures, rendering, etc.), even though most of these terms will be explained again. Moreover, to fully understand the examples, you should have at least read through the SCOL documentation concerning graphic interfaces: to display a 3D image, you need to create a window and a surface object and then copy the surface object into the window. Three functions will be sufficient in the first instance.

The chapter will be divided into four sections. In the first section, we shall review basic 3D concepts. In the second section, scene definition files will be discussed. In the third section, we shall describe the manipulation and rendering functions. In the fourth section, we shall address the tricky issue of collisions.

7.1. Basic 3D concepts

7.1.1. Scene

The scene is the fundamental concept of 3D. A scene is a group of elements such as 3D objects, cameras or collision objects. This group is not organized in a random fashion: it fact, it is arranged as a tree structure. Each node of the tree is a 3D object, a camera, or a collision object. The node which has no parent is called the tree's **root**. The concepts of **parent node**, **child node** and **sibling node** are logically defined.

Geometrically speaking, each node defines a **location**. The location of the root will be called the **global location**. A location is characterized in relation to the parent node's location by **position** and **orientation coordinates**, and by a scale parameter.

3D objects here will be groups of polygons. The polygons will either be triangles or convex quadrilaterals. For example, a cube will be formed of 6 polygons: a square for each face of the cube. On each polygon (or face), you will be able to apply either a uniform color to the face or a texture, i.e., an image. 3D objects will be called '**mesh**'.

There is a special case of 3D object: the empty 3D object, which does not contain any polygons. This object will be called '**shell**'. Thus a 'shell' is used to define an empty location to which you can link different elements. By moving this 'shell', you will move all the attached elements.

Cameras resemble real cameras: in particular, a focal length is defined. Cameras will be called '**camera**'.

Collision objects are groups of spheres, rectangular parallelepipeds, or triangles which define space-filled zones. These modules will be called '**coll**'.

The main purpose of the 3D Voy@ger is to calculate the image that a particular camera "sees". To do this, you need to specify a camera and a surface in which the image will be calculated. The scene thus taken into account will be the scene to which the camera belongs.

7.1.2. Session

The 3D Voy@ger is used to manage several scenes simultaneously. Indeed, the concept of the scene is replaced by the concept of the session. A 3D session is a group of elements such as those described above (mesh, shell, camera, coll). Each element possibly has a parent; there are as many scenes as there are orphan elements.

7.1.3. Material

A material is an element that determines the appearance of a polygon.

A material can be '**flat**'; this means that a uniform color is applied to the face. This color can vary according to the lighting, i.e., according to the face's orientation with respect to the light source. This color can also be translucent: it acts as a color filter. The strength of the filter varies between total transparency and complete opacity.

material can be '**textured**': this means that an image is applied to the face. This image can be a drawing or a photo. The material can be transparent: this means that the dots in a color (which the user can determine) will be considered to be transparent. You can assign a transparency coefficient to the others.

On a textured material, you can apply various colorimetric filters: filters towards a color, color rotation, change in saturation, etc.

The materials are elements of the session. They are not geometric elements: thus they are not attached to locations and, of course, they have no parent.

7.1.4. Performance

It is always difficult to be precise about levels of performance, but we can give the general principles that are used to optimize scenes.

7.1.4.1. Speed of execution

Speed of execution primarily concerns speed of rendering. Other operations (movement, rotation, etc.) are very fast.

- The fewer the polygons, the faster the rendering (avoid exceeding 15,000).
- At a constant number of polygons, rendering is faster if there are lots of meshes in the scene.

- Flat material is much faster than textured material.
- Transparency (both for flat and textured) is particularly memory-hungry, especially when transparent faces are superimposed.

7.1.4.2. Space in memory

The space used in your computer's memory is an important parameter: if the scene is too large, your computer will have insufficient memory and will use *'swap'* mechanisms, which are time-hungry. It is important to distinguish between the size of your 3D files and the space used to store them in the memory. For example, a file containing a *jpeg*-format image with a resolution of 256x256 can have a size of 6 Kb (if it is sufficiently compressed), but will still take up 128 Kb of memory. This latter figure is the important one and is obtained as follows: 256x256x2, since two bytes are needed to store the color of a dot.

The size taken up by information other than textures is negligible: 2 Mb will normally be used to manage scenes of 15,000 polygons.

Above 10 Mb of textures, your scene will have difficulty rotating on a computer that has 32 Mb of random access memory. Therefore, use your judgement.

7.1.5. SCOL 3D Voy@ger characteristics

7.1.5.1. Memory management

The Voy@ger uses a dedicated memory strip to store all its data, except for textures. This memory strip is also used to generate rendering.

The Voy@ger manages a list of materials and a list of textures. Cleaning up is done automatically by means of a "reference counter"-type GC (Garbage Collector).

Memory management is automatic for the following:

- When a material is no longer referenced by an object, it is deleted.
- When a texture is no longer referenced by a material, it is deleted.

In this way, you cannot copy a material without specifying at least one object that uses the copy. Similarly, you cannot copy a texture without specifying at least one material that uses the copy.

Thus there are three types of data in the memory strip:

- The objects which reference any number of materials (at least one per polygon).
- The materials which reference at least one texture.
- The textures which are either loaded (the texture's image is in the memory) or not loaded (the texture's image is not yet in the memory). In the latter case, the materials which reference these textures are automatically rendered as flat.

7.1.5.2. Scene

The objects in the scene are organized as tree structures. The objects are of four types:

- "shell" type: empty object
- "mesh" type: group of points and polygons
- "camera" type: visualization camera
- "coll" type: collision object

The move, create and delete functions are applied to objects in general, without distinguishing type, which simplifies the API.

7.1.5.3. Materials

There are two general types of material: textured materials and untextured materials. Every material has an untextured mode. You can load textures at any time.

Rendering is in 15 bits unpaletted. Textures are in paletted 8-bit **bmp** format (no longer recommended as it is cumbersome to download and of mediocre quality) or in **jpeg** format. You are advised to use jpeg only, since this format offers the best performance in terms of compression.

Textured materials have a transparent mode. For textures in bmp format, the color 0 is the transparency color. For textures in jpeg format, the transparency color can be specified.

Texturing can be done in 'environmental mapping'.

Different filters can be applied:

- Color filter: calculates for each dot the barycenter between the dot's color and the filter's color. The barycenter coefficient is variable.
- Attractor filter: a dot whose color is located at a distance from the color filter below a specific value is forced to the color of the filter.
- Saturation filter: calculates for each dot the barycenter between the dot's saturation and the filter's saturation. The barycenter coefficient is variable.
- Rotation filter: causes a color's shade to be turned from a specific angle.

Materials can use a level of transparency that varies between 0 (opaque) and 255 (transparent). With the **software** Voy@ger, textured materials can have only two levels of transparency:

- from 0 to 127: opaque
- from 128 to 255: 50% transparency

Gouraud shading is available in software mode and hardware mode.

7.2. 3D file format

In order to define a scene, you need to define different elements: meshes, camera, collision objects, materials, etc. It is convenient to have a file format that can be used to describe a scene or part of a scene simply and comprehensively. We will see that the 3D Voy@ger possesses manipulation functions that can subsequently be used to modify virtually any characteristic defined by the scene files. The format of SCOL 3D files is called M3D.

The M3D format is used to describe hierarchically-arranged scenes containing materials, meshes, cameras, and other files to be included. An M3D file is a text file which can be opened using any editor. The file's basic element is the line. The file is divided into blocks of types:

```
type_of element name_of_element {
... (contents)
}
```

The type of element is: material, mesh, shell, camera, coll.

Blocks can be nested (except for 'material' blocks) :

```
shell x {
... (contents)
mesh y {
... (contents)
}
mesh z {
```

```
... (contents)
camera w {
... (contents)
}
}
}
```

Nesting is used to account for a scene's tree structure.

Material blocks are in the form:

```
material name_of_material {
color 123456
texture name_of_texture.jpg
type NOLIGHT
type TRANSPARENCY120
}
```

No line here is indispensable. However, it goes without saying that the minimum you must define is a flat color ('`color`' line) or a texture file ('`texture`' line).

The color is given in 6-digit hexadecimal RGB (8 bits R, 8 bits G, 8 bits B). For example, bright red is coded ff0000.

The texture is a file name which can be preceded by a filter, placed between '%'. The filter is a character string which consists of a list of processes:

- C[6 color characters][2 rating characters]: color filter (expressed as 24 bits hexa) with a rating varying between 0 and 255 (2 characters hexa)
- X: changes colors Red->Blue->Green->Red
- Y: changes colors Red->Green->Blue->Red
- A[6 color characters][2 distance characters]: attractor. Every dot close to the color expressed is forced to this color. This color becomes the transparency color for jpeg textures.
- R[4 rotation characters]: shade rotation (Hsv model). The angle is between 0 and 65535.
- S[4 value characters][2 rate characters]: saturation filter. The restricted saturation value is given on 4 bytes and is between 0 and 65536. The rating varies between 0 and 255.

Example : `%Cd0000080X%toto.jpg`

In this example, a red filter at 50% followed by a color rotation X will be applied to the texture toto.jpg.

The lines of types are: NOLIGHT, TRANSPARENCY120. You can put in as many type lines as you wish. The coefficient which follows the word TRANSPARENCY is between 0 (opaque) and 255 (transparent). It is only useful during an untextured rendering.

The mesh blocks are in the form:

```
mesh name_of_mesh {
x y z a b c scale
vertices
[light l]
polygons
[recursion]
}
```

The mesh's position is given by the coordinates `x, y, z` and the angles `a, b, c` (between 0 and 65535). The `scale` parameter is optional. It is expressed as a percentage: 100 represents the normal scale, 200 for double-size, 50 for half-size, etc.

The vertices are lines of three coordinates `x, y, z,` which are whole or floating integers.

Light `l` is optional and should be between 0 (very dark) and 31 (value used by default).

The polygons are described in the form of blocks comprising:

- a line giving the name of the material to be applied to the polygons to follow
- lines of polygons comprising 3, 4, 6, 8, 9 or 12 integers, according to whether you define 3 or 4 apices, with 0, 1 or 2 texture coordinates.

Following the polygons are the means to insert other meshes, cameras..

The camera blocks are in the form:

```
camera name_of_camera {
x y z a b c scale
distx disty sx sy
zclip zfog zback
}
```

The camera's position is given by the coordinates `x, y, z` and angles `a, b, c` (between 0 and 65535). The `scale` parameter is optional.

The `distx` and `disty` parameters give the distance from the screen on the x-axis and on the y-axis (used in the projection formulas `X=distx*x/z` et `Y=disty*y/z`).

The `sx` and `sy` parameters give the screen's half-width and half-height.

Shell refers to an empty object (neither camera nor mesh) which is used only for attaching other objects to it. The syntax of a **Shell** object is as follows:

```
shell name_of_shell {
x y z a b c scale
}
```

Following the **shell** coordinates are the means to insert other meshes, cameras or to include files.

The # character indicates that what follows the line is a comment.

Example :

```
# cube
material wood {
 color c04000
 texture wood.jpg
}

material stone {
 color 00c040
 texture stone.jpg
 type TRANSPARENCY
}

mesh cube {
0 0 0  0 0 0

-100 -100 -100
100 -100 -100
100 100 -100
-100 100 -100
-100 -100 100
```

```
100 -100 100
100 100 100
-100 100 100

light 10
wood
 0 0 0   1 255 0   2 255 255 3 0 255
 1 0 0   5 255 0   6 255 255 2 0 255
 5 0 0   4 255 0   7 255 255 6 0 255
stone
 4 0 0   0 255 3   3 255 255 7 0 255
 3 0 0   2 255 0   6 255 255 7 0 255
 1 0 0   0 255 0   4 255 255 5 0 255
}
```

In this example, we have defined a cube with an edge of 200. There are three faces which use the material 'wood' and three faces which use the material 'stone'.

7.3. 3D manipulation API

The basic example includes the following points:

- creating a bitmap in which rendering will be performed
- creating a session
- reading an M3D file containing a cube and a camera
- calculating rendering
- displaying the bitmap in a window.

We will use three files to do this:

file '*Tutorial/mytest3d.scol*'

```
_load "Tutorial/mytest3d.pkg"
main
```

file '*Tutorial/mytest3d.pkg*'

```
/* MyTest 3d */
typeof win=ObjWin;;
typeof buffer=ObjSurface;;
typeof session=S3d;;
typeof shell=H3d;;
typeof camera=H3d;;

fun _end(a,b)=_closemachine;;
fun _paint(a,b)=_BLTsurface win 0 0 buffer 0 0 400 300;;

fun main()=
 set win=_CRwindow _channel nil 150 150 400 300
  WN_MENU|WN_MINBOX "My 3d Test";
 _CBwinDestroy win @_end nil;
 _CBwinPaint win @_paint nil;
 set buffer=_CRsurface _channel 400 300;
 set session = MX3create _channel 1024 1024 1024 1024*1024;
 if session==nil then _closemachine
 else
 (set shell = M3createShell session;
  M3load session "Tutorial/scene.m3d" shell;
```

```
  set camera=M3getObj session "camera";
  M3recursFillMatObj session shell;
  MX3render session buffer camera 0 0 0;
  _paint nil nil;
  0);;
```

file *'Tutorial/scene.m3d'*

```
# cube
material wood {
 color c04000
 texture Tutorial/wood.jpg
}

material stone {
 color 00c040
 texture Tutorial/stone.jpg
}

mesh cube {
0 0 0  0 0 0

-100 -100 -100
100 -100 -100
100 100 -100
-100 100 -100
-100 -100 100
100 -100 100
100 100 100
-100 100 100

wood
 0 0 0   1 255 0   2 255 255 3 0 255
 1 0 0   5 255 0   6 255 255 2 0 255
 5 0 0   4 255 0   7 255 255 6 0 255
stone
 4 0 0   0 255 3   3 255 255 7 0 255
 3 0 0   2 255 0   6 255 255 7 0 255
 1 0 0   0 255 0   4 255 255 5 0 255

camera camera {
200 300 -400 7000 -5000 0
200 200 200 150
10 10000 10000
}
}
```

If you try the example, you will obtain a window containing a cube seen three-quarters on. The faces are colored. To obtain textures, you must create two graphic files, *'Tutorial/stone.jpg'* and *'Tutorial/wood.jpg'*. You can also rename these files at the start of the *Tutorial/scene.m3d* file and replace them with jpeg files.

In the example, we notice a few new functions whose name begins with '`M3`' or '`MX3`': these are SCOL 3D Voy@ger API functions. Below we outline these functions before moving on to give an exhaustive list.

`set session = MX3create _channel 1024 1024 1024 1024 1024*1024;`
Creates a 3D session whose size in the memory strip is 1 Mb. The session variable has the type 'S3d'.

`(set shell = M3createShell session;`
Creates a shell element in the session. The `shell` variable has the type 'H3d'.

`M3load session "Tutorial/scene.m3d" shell;`
Reads the *Tutorial/scene.m3d* file and puts its entire contents under the shell element.

`set camera=M3getObj session "camera";`
Finds the trace of the object called 'camera' in the scene. The camera variable has the type 'H3d'.

`M3recursFillMatObj session shell;`
Loads all the scene's textures, or, more exactly, the textures used by the objects located under the shell object.

`MX3render session buffer camera 0 0 0;`
Calculates rendering of what the camera 'sees' in the buffer.

Note: functions that start with MX3 have been introduced to support 3D cards and pass in software mode if there is no 3D card. Thus the older functions M3create and M3scanline are now obsolete.

7.3.1. New types

We define the following magma types:

- `S3d` 3D session
- `H3d` 3D object handler
- `Hmat3d` material handler
- `Htx3d` texture handler

Generally speaking, the functions return 0 if successful.

7.3.2. Session

`S3d MX3create(channel Chn, nbmat I, nbtext I, nbobj I, ntaby I, sizetape I)`
Initializes a 3D session by specifying the maximum number of materials, textures, objects and lines of rendering and the size of the memory strip to be allocated for storing the meshes and calculating rendering.

`I M3destroy(session H3d)`
Destroys the 3D session.

`I M3freeMemory (session S3d)`
Returns the amount of memory still free in the session. Performs a GC.

`I M3reset (session S3d)`
Deletes all the objects and textures in the session.

7.3.3. General object management

`I M3load(session S3d, file S, parent H3d)`
Loads a file in M3D format by specifying the parent handler (`nil` if none). If the parent is specified, the file elements will be considered as its child elements.

I M3loadString(session S3d, contents S, parent H3d)
Loads an object from a character string in M3D format by specifying the parent handler (`nil` if none).

H3d M3createShell(session S3d, parent H3d)
Creates a Shell-type object and returns its handler (`nil` in the event of failure).

I M3delObj(session S3d, object H3d)
Destroys an object by knowing its handler.

H3d M3copyObj(session S3d, object H3d)
Creates a copy of the object. This copy is not attached to any tree structure.

H3d M3getObj(session S3d, name S)
Gives an object handler depending on its name; nil if it cannot be found.

S M3objName(session S3d, object H3d)
Returns the object's name.

H3d M3getFather(session S3d, object H3d)
Returns the handler of an object's parent.

H3d M3getFirstSon(session S3d, object H3d)
Returns the handler of an object's first child.

H3d M3getBrother(session S3d, object H3d)
Returns the handler of an object's sibling.

H3d M3bigFather(session S3d, object H3d)
Returns the handler of the apex of the tree to which an object belongs.

H3d M3isFather(session S3d, child H3d, parent H3d)
Returns 1 if parent is well above child in the tree structure.

I M3unLink(session S3d, parent H3d)
Detaches an object (and all its descendants). The object remains in the memory.

I M3link(session S3d, child H3d, parent H3d)
Attaches one object beneath another.

I M3renameObj(session S3d, object H3d, name S)
Changes an object's name (must be less than 32 characters).

I M3setObjVec(session S3d, object H3d ,vector [I I I])
Defines an object's position in the location of its parent.

[I I I] M3getObjVec(session S3d, object H3d)
Reads an object's position in the location of its parent.

I M3setObjAng(session S3d, object H3d, angular [I I I])
Defines an object's angular position.

[I I I] M3getObjAng(session S3d, object H3d)
Reads an object's angular position.

I M3setObjScale(session S3d, object H3d, scale I)
Defines an object's scale, expressed as a percentage: 100 for normal size, 200 for double-size, etc.

I M3getObjScale(session S3d, object H3d)
Reads an object's scale, expressed as a percentage: 100 for normal size, 200 for double-size, etc.

I M3movObj(session S3d, object H3d, vector [I I I])
Moves an object from a vector in the object's frame of reference. For a camera, the vision axis is z, the horizontal axis is x, and the vertical axis is y.

I M3rotateObj(session S3d, object H3d, angular [I I I])
Turns an object from an angular vector in the object's frame of reference.

I M3movObjExt(session S3d, object H3d, vector [I I I])
Moves an object from a vector in the frame of reference of the object's parent.

I M3rotateObjExt(session S3d, object H3d, angular [I I I])
Turns an object from an angular vector in the frame of reference of the object's parent.

I M3calcMat(session S3d, object H3d);
Calculates the positions of objects in a scene in the object's location. You can retrieve these positions with the following two functions.

[I I I] M3getObjVecRender(session S3d, object H3d)
Gives an object's position after `M3calcMat or M3scanline`.

[[I I I] [I I I] [I I I]] M3getObjMatrixRender(session S3d, object H3d)
Gives an object's matrix after `M3calcMat or M3scanline`. The matrix consists of 32-bit integers, with 16 bits after the comma.

[[I I] [I I] [I I I]] M3getCamera(session S3d, object H3d)
Gives a camera's parameters: `[ [dx dy] [sx sy] [zclip zmiddle zmax] ]:`

- `dx, dy`: projection distance (formula X=dx.x/z, Y=dy.y/z)
- `sx, sy`: screen's half-width and half-height
- `zclip, zmiddle, zmax`: start of fog near clipping distance and far clipping distance.

I M3setCamera(session S3d, object H3d, parameters [[I I] [I I] [I I I]])
Sets a camera's parameters: `[ [dx dy] [sx sy] [zclip zmiddle zmax] ].`

[H3d r1] M3ListOfBigFathers (session S3d)
Returns a list of all objects without a parent, i.e., those which are at the apex of a hierarchy.

[[I I I] [[I I I] [I I I] [I I I]]] M3calcPosRef (session S3d, object H3d, reference H3d)
Calculates the object's position in the location of the reference object. The matrix consists of 32-bit integers, with 16 bits after the comma.

[I I I] M3angularFromMatrix (matrix [[I I I] [I I I] [I I I]])
Calculates a triplet of angles corresponding to the pass matrix. The matrix consists of 32-bit integers, with 16 bits after the comma.

I M3getRadius (session S3d, object H3d)

Reads the object's radius (maximum distance from the center of the object's location to its vertices).

`[x I y I z I r I] M3calcProj (session S3d, camera H3d, object H3d)`
Calculates an object's projection on a camera and returns, if the object is visible, the screen x and y coordinates, the distance z and the apparent radius r. Returns `nil` if the object is not visible or in the event of an error. This calculation only takes account of the z clipping (front and back). It does not take account of lateral clippings.

`[I I I] M3angularTarget (src_vector [I I I], dest_vector [I I I])`
Returns the angular positions that enable the source to point towards the destination. The third angle is always nil.

`I M3getObjType (session S3d, object H3d)`
Returns the object's type:

- M3_SHELL
- M3_CAM
- M3_MESH
- M3_COLL
- M3_LIGHT

`[I I I] M3getGlobalVec (session S3d, location H3d, vector [I I I])`
Gives the global coordinates of a vector expressed in the location. By global coordinates we mean coordinates in the location situated at the apex of the tree to which the location passed as a parameter belongs.

7.3.4. Managing materials

`HMat3d M3getMat(session S3d, name S)`
Returns the handle of a material depending on its name; `nil` if it cannot be found.

`S M3materialName(session S3d, material HMat3d)`
Returns the name of the material.

`I M3renameMat(session S3d, material HMat3d, name S)`
Changes the name of a material.

`I M3fillMat(session S3d, material HMat3d)`
Loads, if necessary, the texture of a material.

`I M3fillMatObj(session S3d, object H3d)`
Tries to load all the textures that are useful to an object.

`I M3recursFillMatObj(session S3d, object H3d)`
Tries to load all the textures that are useful to an object and its descendants. If the object is the apex of the scene, tries to load all the textures in the scene.

`I M3getType(session S3d, material HMat3d)`
Returns the default type of a material (i.e., as designated by the user).
The result is a component of the following masks:

- MAT_TEXTURED: textured material
- MAT_TRANSP: material with transparency effect

- MAT_LIGHT: material with lighting effect
- MAT_ENV: environmental mapping
- MAT_GOURAUD: Gouraud shading

`I M3getRealType(session S3d, material HMat3d)`
Returns the current type of a material (default type, may be altered: e.g., if the map is not loaded, the current type of a material is forced to flat). See above for masks.

`I M3setType(session S3d, material HMat3d, type I)`
Defines the default type of a material, taking account of constraints. See above for masks.

`I M3getMaterialFlat(session S3d, material HMat3d)`
Returns the color associated with the material.

`I M3setMaterialFlat(session S3d, material HMat3d, color I)`
Changes the color associated with the material.

`I M3getMaterialTransparency(session S3d, material HMat3d)`
Returns the transparency coefficient associated with the material (value between 0 and 255).

`I M3setMaterialTransparency(session S3d, material HMat3d, coef I)`
Changes the transparency coefficient associated with the material (value between 0 and 255). Flat rendering only used.

`I M3materialCount (session S3d, material HMat3d)`
Returns the reference counter associated with the material.

`[HMat3d r1] M3listOfMaterials(session S3d, object H3d)`
Returns the list of materials used by the object.

`HMat3d M3copyObjMaterial (session S3d, object H3d, material HMat3d)`
Creates a new material as a replacement and copy of another material in an object. Returns the new material.

`I M3chgObjMaterial (session S3d, object H3d, current_material HMat3d, new_material HMat3d)`
Replaces a material with another one in a specified object.

7.3.5. Managing textures

`HTx3d M3getTexture(session S3d, name S)`
Returns the handle of a texture depending on its name; `nil` if it cannot be found.

`S M3textureName(session S3d, texture HTx3d)`
Returns the texture's name.

`I M3renameTexture(session S3d, texture HTx3d, name S)`
Changes a texture's name.

`HTx3d M3textureFromMaterial (session S3d, material HMat3d)`
Returns the texture associated with the material.

`I M3textureCount (session S3d, texture HTx3d)`

Returns the reference counter associated with the texture.

`I M3shiftTextureXY(session S3d, object H3d, material HMat3d, x I, y I)`
Shifts the mapping coordinates of the polygons of the object using a particular material.

`HTx3d M3copyMaterialTexture (session S3d, material HMat3d)`
Creates a new texture as a replacement and copy of another texture in a particular material. Returns the new texture.

`I M3chgMaterialTexture (session S3d, material HMat3d, new_texture HTx3d)`
Replaces the texture of a material with another texture.

`I M3isTextureFilled (session S3d, texture HTx3d)`
Returns 1 if a texture's bitmap is loaded, 0 if not.

`I M3fillTexture (session S3d, texture HTx3d)`
Loads a texture's bitmap.

`I M3freeTexture (session S3d, texture HTx3d)`
Frees a texture's bitmap.

7.3.6. Managing rendering and link with 2D interface

`H3d MX3render(session S3d, buffer ObjSurface, objet H3d, xplot I, yplot I, col I)`
Calculates rendering of a scene from the 'object' handler camera in a rendering buffer. You specify the background color col (`nil` for transparent). The function returns the handler of the object located at the point of the (`xplot,yplot`) coordinates, nil if none. NB: the camera's rendering size should not exceed the buffer bitmap size. A camera's rendering size is double the camera's half-heights and half-widths. Use M3setCamera to modify these values.

`[H3d HMat3d] MX3renderm(session S3d, buffer ObjSurface, object H3d, xplot I, yplot I, col I)`
Calculates rendering of a scene from the ic handler camera in a rendering buffer. You specify the background color col (`nil` for transparent). The function returns the handler and the material of the object located at the point of the (`xplot,yplot`) coordinates, nil if none.

`[H3d HMat3d I I I] MX3renderInfo(session S3d, object H3d, xplot I, yplot I)`
Returns information on the (`xplot, yplot`) point in the following order:

- 3D handler
- material handler
- texture u coordinate
- texture v coordinate.

`I M3blitTexture (session S3d, texture HTx3d, buffer ObjBitmap8)`
Replaces the contents of a texture with another (in 8 bits) by erasing the palette.

`I M3blitTexture16 (session S3d, texture HTx3d, buffer ObjBitmap)`
Replaces the contents of a texture with another (in 16 bits) by erasing the palette.

`ObjBitmap M3filter (buffer ObjBitmap, filter S)`
Applies a filter to a bitmap. This filter is identical to the one used in loading textures.

Example: red filter 1/8: Cff000020

7.4. Managing collisions

7.4.1. Principles

7.4.1.1. Collision elements

The 3D Voy@ger of the SCOL machine contains a collision management system. This system is based on **collision elements**, which the user can define in a number of ways: spheres, boxes, or polygons. Boxes will in fact be **'Obb'** (Oriented Bounding Boxes): an Obb is any rectangular parallelepiped. Its axes are not necessarily the scene's axes.

These collision elements have no direct relation with the scene's mesh elements. In fact, it is important to be able to define invisible collision elements (e.g. invisible walls at the edge of a link in empty space to prevent users from falling into the void) as well as mesh elements which will not be taken into account in calculating collisions, either because the user has to be able to pass through these elements, or in the interests of optimization (perhaps users cannot be located near the element in any way, or you may want to replace complicated geometry with a simple, all-encompassing box).

An intersection and collision detection system is not only applicable to testing the position of a camera in relation to a set. In fact, it is also generally applicable to the testing of two groups of collision elements.

From the collision elements, the Voy@ger is used to detect two types of event:

- intersection: do two groups of collision elements A and B intersect?
- collision in translation: given a movement vector ***u*** and two groups of collision elements A and B, will there be intersection between A and B when A is moved according to vector ***u***? If so, at what moment during the movement and according to which plane will the collision take place?

7.4.1.2. Integrating collision elements into the scene

We shall distinguish between **primary** collision elements and **secondary** collision elements.

A primary collision element is the basic collision element: these are the spheres, boxes, and polygons considered as the scene's solid elements, with which you carry out intersection and collision tests.

Some primary collision elements can be grouped in binary tree structures (each node has at most two child elements). In fact, the primary collision elements are located in the leaves of a tree. The nodes which are not leaves are secondary collision elements. They are of the same kind (sphere, box, or polygon), but they are not real collision elements: they incorporate primary collision elements and are used to optimize calculation. The algorithm (of intersection or collision) is derived from the principle that if there is intersection or collision with the parent, then there can be intersection or collision with the child. Conversely, if there is no intersection or collision with the parent, then there cannot be intersection or collision with the child.

In each node of the tree we find not a (primary or secondary) collision element but a list of collision elements of the same kind (spheres, boxes, or polygons). The algorithm moreover supposes that if a child is of a different type from its parent (for example, the child is OBB type and the parent Sphere type), then the child is an only child.

The scene's collision elements are in fact trees of this kind, which may be reduced to a single leaf containing a single collision element. You can thus define as many collision objects as you wish in a scene. These objects, like others, are defined in relation to the location of their parent and may

themselves have child objects. For example, you can define an OBB-type collision object having the size of an object (e.g. a table). The collision object will be defined as a child of the table, which simply means that the table only has to be moved for the collision object to follow.

The intersection or collision algorithm takes at the start two of the scene's nodes; let us call them A and B. It carries out its calculation taking into account all the collision objects located under A and all the collision objects located under B. If A belongs to a descendant of B, the calculation will be performed between the objects located under A and the objects located under B except for those located under A. Usually, you can provide the algorithm with the object that is moving and the whole of the scene.

7.4.1.3. Managing collisions

It is one thing to detect an intersection or a collision, and quite another to manage this kind of event, in other words, to propose a movement, more generally a transformation, that will allow the scene to quit this state of intersection or collision. This problem is particularly complex since it is specific to the application and requires the selection of a "physical" model for the scene. This is not purely a geometric problem. The 3D Voy@ger, however, provides help in resolving collisions. The `M3testColl` collision function between A and B returns supplementary information connected to vector ***u***:

- the movement that is geometrically possible: a vector $\lambda.u$ with λ between 0 and 1.
- the collision axis ***v***, such that ***u+v*** is a vector that transports object A to a guard distance from B in such a way as to cause A to slide along B. In doing so, A may enter into collision with another object C, or even with another part of B.

7.4.2. API

`H3d M3createSphere(session S3d, radius I)`
Creates a collision node containing a sphere whose radius is given as a parameter.

`H3d M3createObb(session S3d, sizes [I I I])`
Creates a collision node containing a sphere and an OBB (the OBB being the child of the sphere). The sizes are the half-sizes of the OBB.

`[[I I I] [I I I]] M3calcObb (session S3d, object H3d)`
Calculates the position and sizes of the OBB covering an object.

`I M3firstRadius (session S3d, object H3d)`
If the object is a collision node whose apex is a sphere, this function returns its radius.

`[H3d H3d] M3testInter (session S3d, objectA H3d, objectB H3d)`
Intersection test between subtrees A and B. If A (resp. B) is a subtree of B (resp. A), the test is carried out between subtree A (resp. B) and subtree B (resp. A) without subtree A (resp. B). Returns `nil` if no intersection, otherwise returns both objects in intersection.

`[H3d H3d [I I I] [I I I]] M3testColl (session S3d, objectA H3d, objectB H3d, vector [I I I], guard I)`
Collision test between subtrees A and B. If A (resp. B) is a subtree of B (resp. A), the test is carried out between subtree A (resp. B) and subtree B (resp. A) without subtree A (resp. B). The collision test is carried out on A's movement according to a vector passed as a parameter, and expressed in the global location. Returns `nil` if no collision, otherwise returns both objects in collision and two vectors:

- the first is collinear to the movement vector and gives the movement up to the collision
- the second is:

- nil if A and B are already in intersection before the movement
- otherwise a perpendicular vector in the plane of collision, such that, added to the movement vector passed as a parameter, a vector is obtained which suggests resolution of the collision.

8. BIGNUM PROGRAMMING

8.1. General introduction

The '*BigNum'* library is used for handling big whole numbers (up to 128 bits).

It offers a variety of operations such as addition, subtraction, Euclidean division, multiplication, exponentials, etc.

The library provides conversion functions between character strings and BigNums, which, among other things, allows these numbers to be saved. The library is also used to convert a complete text to a list of '*BigNum'*, and vice versa.

8.2. API

The API defines a new type: **BigN**

Conversion functions:

`BigFromAsc [string S] BigN`
This function is used to create a BigNum from a character string coded in hexadecimal.
Example:

- BigFromAsc "1" : defines the number 1
- BigFromAsc "f " **:** defines the number 15

`BigToAsc : fun [ number BigN ] S`
Opposite function to the previous function.

`BigToString : fun [ number BigN ] S`
Converts a bignum into any character string (the smallest being able to contain the number). Internal coding has not been specified here; this function is used only for handling bignums in communications or in files. The size of the string produced is approximately two times smaller than the `BigToAsc` function ASCII string.

`BigToStringn : fun [ number BigN, size I ] S`
Same function as above, but specifying the size of the string to be produced. If it is too short, the BigNum integer will be truncated.

`BigFromString : fun [string S] BigN`
Opposite function to `BigToString`.

`BigRand : fun [ ] BigN`
Randomly chooses a number of 127 bits. Randomness can be manipulated by the srand function in the standard library.

`BigPrimal : fun [ i I ] BigN`
Randomly chooses a primary number of i bits in the form 3n+2. Randomness can be manipulated by the srand function in the standard library.

BigAdd : fun [x BigN, y BigN] BigN
Performs addition: `x+y`.

BigSub : fun [x BigN, y BigN] BigN
Performs subtraction: `x-y`.

BigXor : fun [x BigN, y BigN] BigN
Performs the exclusive or operation: `x^y`.

BigMod : fun [x BigN, n BigN] BigN
Performs the operation: `x mod n`

BigDiv : fun [x BigN, y BigN] BigN
Performs Euclidean division: `x div y`

BigMul : fun [x BigN, y BigN] BigN
Performs multiplication (modulo 2^127): `x*y`

BigMuln : fun [x BigN, y BigN, n BigN] BigN
Performs multiplication: `x*y mod n`

BigInvn : fun [x BigN, n BigN] BigN
Calculates the inverse (Bezout's theorem): `1/x mod n`

BigExpn : fun [x BigN, y BigN, n BigN] BigN
Calculates the exponential: x^{y} mod n

BigPgcd : fun [x BigN, y BigN] BigN
Calculates the pgcd of x and y.

BigCmp : fun [x BigN, y BigN] I
Compares unsigned integers x and y. Returns:

- 0 if x=y
- 1 if x>y
- -1 if x<y

BigListFromString : fun [text S, size_block I] [BigN r1]
Divides a text into a list of words each of size_block bytes. Each word is then converted into a BigNum. The function randomly completes the end of the message to attain a multiple size_block size.

BigListToString : fun [liste [BigN r1], size_bloc I] S
Opposite function to the above: reconstitutes the original text, on condition that the same value of size_block is used.

8.3. Example

The following example shows how to code the RSA algorithm. Let us suppose that the public exponent is 3.

The three API functions are:

- RSAcreate3: calculates a public key-private key pair.
- RSAcrypt3: encrypts a text using the public key.
- RSAdecrypt: decrypts a text using the private key.

NB: check that you have the necessary legal authorization before attempting to integrate this example in one of your applications.

```
/*
 RSA library - jul 97 -
*/

/* functions for internal use */
fun RSAsizebis(n,b,i)=
 if (BigCmp b n) >0 then i
 else RSAsizebis n BigAdd b b i+1;;

fun RSAsize(n)= RSAsizebis n BigFromAsc "1" 0;;

fun RSAcryptone3(l,n,i1)=
 if l==nil then nil else
 let l->[m nxt] in
   (BigToStringn (BigMuln (BigMuln m m n) m n) i1)::RSAcryptone3 nxt n i1;;

fun RSAcutstring(s,i,n)=
 let substr s i n -> z in
 if strlen z then z::RSAcutstring s i+n n
 else nil;;

fun RSAdecryptone(l,k,n)=
 if l==nil then nil else
 let l->[m nxt] in
  (BigExpn BigFromString m k n)::RSAdecryptone nxt k n;;

/* API */
/* decrypting a message s with the private key n,k -> returns a string */
fun RSAdecrypt(s,k,n)=
 let ((RSAsize n)-1)>>3 -> nbyte in
 BigListToString (RSAdecryptone (RSAcutstring s 0 nbyte+1) k n) nbyte;;

/* encrypting a message s with the public key n -> returns a string */
fun RSAcrypt3(s,n)=
 let ((RSAsize n)-1)>>3 -> nbyte in
 strcatn RSAcryptone3 (BigListFromString s nbyte) n nbyte+1;;

/* creating a public key-private key pair of approx i bits (between i and
i-1)
  returns the pair [n k]*/
fun RSAcreate3(i)=
 let [BigFromAsc "1" BigFromAsc "3"] -> [one three] in
 let [BigPrimal i>>1 BigPrimal i-(i>>1)] ->[p q] in
 if  ! BigCmp p q then RSAcreate3 i
 else let BigMul p q ->n in
 let BigMul BigSub p un BigSub q un -> phi in
 let BigInvn trois phi -> k in
  [n k];;
```

9. SQL

9.1. General introduction

The SQL library is used for handling any database using the *ODBC* support and SQL queries. The SQL library is not supplied in the basic SCOL Voy@ger since it requires the ODBC support, which is not found on every machine. This library is supplied as a SCOL plug-in, **"scolsql.dll"**, and the following line must appear in the **usm.ini** file:

```
plugin plugins/scolsql.dll SCOLloadSQL
```

In order for it to operate, the database must be defined and named in the Windows configuration panel (menu odbc32bits). The name then given to the pair (database file/associated driver) will be the database's determinant, enabling SCOL to access it.

The API has 3 functions and 2 types.

9.2. API

The API defines a new type: **SqlDB**

This type corresponds to a database connection.

9.2.1. Connection creation function:

```
SqlDB SqlCreate (Channel Chn, Database_name S, Login S, Password S)
```

This function returns `nil` if the connection fails. The name of the database is that defined in the configuration panel, *'menu odbc32bits'*. The login and '*password'* are authentication parameters used to connect to the database. If you have not defined access rights to your database, the "admin" login and the "password" should work.

9.2.2. Disconnection function:

```
I SqlDestroy (connection SqlDB)
```

This function closes the connection.

9.2.3. Request function:

```
[[S r1]r1] SqlRequest (connection SqlDB, SQL_request S, parameters [SqlParam r1])
```

The SQL request is in text format. The request's parameters appear in the form of the `'?'` character. The list of parameters (the `SqlRequest` function's third argument) respects the order and number of the `'?'` in the SQL query.

The result is a list of character strings corresponding to the list of responses to the request, where each response is a list of columns.

The parameters are passed as a list of **SqlParam**-type parameters. The **SqlParam** type is defined as follows:

```
typedef SqlParam =
 SQL_BIGINT S | SQL_BINARY S | SQL_BIT S | SQL_CHAR S |
 SQL_DATE S | SQL_DECIMAL S | SQL_DOUBLE S | SQL_FLOAT S |
 SQL_INTEGER S | SQL_LONGVARBINARY S | SQL_LONGVARCHAR S | SQL_NUMERIC S |
 SQL_REAL S | SQL_SMALLINT S | SQL_TIME S | SQL_TIMESTAMP S |
 SQL_TINYINT S | SQL_VARBINARY S | SQL_VARCHAR
```

9.3. Examples

The following example opens the *"MyBase"* database and performs a password search as a function of a "Foo" *login*.

```
typeof db=SqlDB;;

fun main()=
set db=SqlCreate _channel "second" "admin" "";
SqlRequest db "SELECT Password FROM Table WHERE Table.Login=?;"
  (SQL_CHAR "Foo")::nil;
SqlDestroy db;
0;;
```

The following example opens the "*MyBase*" database and performs a login search on the basis of a telephone number "1234".

```
typeof db=SqlDB;;

fun main()=
set db=SqlCreate _channel "second" "admin" "";
SqlRequest db "SELECT Login FROM Table WHERE Table.Tel=?;"
(SQL_NUMERIC "123")::nil;
SqlDestroy db;
0;;
```

The following example opens the "*MyBase*" database and outputs the list of logins and telephone numbers on the *(_fooS strbuild)* console.

```
typeof db=SqlDB;;

fun main()=
set db=SqlCreate _channel "second" "admin" "";
_fooS strbuild SqlRequest db "SELECT Login,Tel FROM Table;" nil;


SqlDestroy db;
0;;
```

The following example adds a new line in the database.

```
typeof db=SqlDB;;

fun main()=
set db=SqlCreate _channel "second" "admin" "";
SqlRequest db "INSERT INTO Table VALUES(?,?,?);"
  (SQL_CHAR "Titi")::(SQL_CHAR "xyz")::(SQL_NUMERIC "789")::nil;
SqlDestroy db;
0;;
```

The following example modifies Titi's password.

```
typeof db=SqlDB;;

fun main()=
set db=SqlCreate _channel "second" "admin" "";
SqlRequest db "UPDATE Table SET Password=? WHERE Login=?;"
  (SQL_CHAR "123abc")::(SQL_CHAR "Titi")::nil;

SqlDestroy db;
0;;
```

The following example deletes Titi's form.

```
typeof db=SqlDB;;

fun main()=
set db=SqlCreate _channel "second" "admin" "";
SqlRequest db "DELETE FROM Logins WHERE Login=?;"
  (SQL_CHAR "Titi")::nil;
SqlDestroy db;
0;;
```

10. HTTP INTERFACING

The SCOL machine integrates the http protocol both on the server side and on the client side. This means that, in a few lines of SCOL, you can not only create a small http server but also make http requests as a client to any http server in the world. Similarly, two SCOL machines can communicate via http, one acting as server, the other as client. This is extremely useful for connecting to a SCOL site when you yourself are behind a firewall which only allows the http protocol to pass.

Below we describe both **server** and **client** aspects.

10.1. Http server

10.1.1. Principles

We define two types :

- **HTTPserver**: Http server
- **HTTPcon**: Http connection. When a request is received by an http server, it creates a "connection" which enables the server program to respond to the request.

To define an http server, you need to:

- choose a free TCP/IP port
- define a callback function, which will be called every time a request is received by the server

You call the function:
```
startHTTPserver : fun [Chn I fun [HTTPcon u0 S] S u0] HTTPserver
```

For example :
```
startHTTPserver _channel 8080 @callback nil
```

The callback function is called when the server has received the complete request. It takes three arguments:

- the http connection
- the user parameter
- a character string containing the request

It must return a character string which is the response to be transmitted. This is synchronous mode: the callback function immediately returns the response to be transmitted.

There is an asynchronous mode, which allows the server to postpone its response. For this the callback function needs to return `'nil'`. The server program can then:

- send the response, packet by packet, using the `HTTPsend` function
- close the connection after the last packet, using the `closeHTTPcon` function
- it can also send the contents of a file, which will be transferred directly from the disk to the http connection without passing through the SCOL machine's memory. At the end of the transfer, the http connection will be automatically closed. This is done with the `HTTPsendFile` function.

10.1.2. *Some words of advice*

An http request contains:

- a header whose first line is in the form VERB URL, the verb generally being GET or POST
- a header ending: `"\13\10\13\10"` (two returns to the line)
- possibly a message body, used for `POST` requests

The simplest way of parsing such a request is to use a `strfind` to find the header ending and to apply the `strextr` function to the header.

The response to an http request must also begin with a header. For example, you can take the following header for a text message:

```
HTTP/1.0 200 OK\13\10Server: SCOL HTTP server\13\10Content-Type:
text/html\13\10\13\10
```

People often forget to send this header, so make sure you don't waste time by making such a careless mistake.

Example:

```
var http_header="HTTP/1.0 200 OK\13\10Server: SCOL HTTP
server\13\10Content-Type: text/html\13\10\13\10";;

fun http_onrequest(con,db,req)=
  let hd strextr req -> l in
  let l->[com [url _]] in
  if (!strcmpi com "GET") then strcat http_header "GET"
  else if (!strcmpi com "POST") then strcat http_header "POST"
  else "";;

...
 startHTTPserver _channel 8080 @http_onrequest nil;
...
```

In this example, you return a page depending on the verb in the request. You will note how it has been parsed. If the verb is neither GET nor POST, you return an empty string. The client will therefore receive this empty string, without a header, which will cause an error, which may be deliberate, as here.

10.2. Http client

10.2.1. *Principles*

To make an http request, the client must specify the verb, the URL, and possibly a message body.

In the case of a GET, only the URL needs to be specified, hence a simplified API in this case. You will use the following function:

```
INETGetURL : fun [Chn S I fun [INET u0 S I] u1 u0] INET
```

The arguments are:

- the proprietary channel
- the URL
- a flag (leave at 0)
- a callback

- a user parameter

The function returns an '**INET**' type corresponding to a request in progress, which can be interrupted if you no longer want to receive the response.

The simplest solution in the majority of cases is to call the callback only after the response has been fully received. However, there are some Internet applications (particularly pseudo-streaming) which return the response in several chunks: several minutes can elapse between the start and end of the response. For this reason, we considered it preferable to inform the SCOL program every time data has been received. The callback thus takes the following 4 arguments:

- `INET` request
- user parameter
- data received
- state of the request:
  - 0: data has been received, it is in the third argument
  - 1: this is the end of the request (the third argument is `nil`)
  - 2: network error (the third argument is also `nil`)

Example :

```
fun cbgethttp(inet,z,s,reason)=
 let z->[content] in
  if reason==0 then (mutate z <- [strcat content s];nil)
  else if reason==1 then (_fooS content; _fooS "OK");
  else _fooS "ERROR";;

...
INETGetURL _channel "http://www.cryo-networks.com" 0 @cbgethttp [nil];
...
```

It is difficult to detect network errors. Indeed, when an http client is behind a proxy and requests a non-existent URL, the client in effect requests the proxy for this URL, and the proxy is then responsible for requesting it from the Internet server corresponding to the URL. If the page does not exist, the proxy will be informed. The proxy then often creates a quite valid html page with a few lines of text explaining that an error has occurred. However, for the http client, the response is in order, not an error message.

If you are developing an application in which you are managing the http client and the http server, you are advised to ensure that clients can easily determine whether the page they received was indeed the one they expected, for example by having a 'magic number' precede it.

10.2.2. POST method

Having just seen how to perform a GET request, let us now move on to the POST method. This enables you to pass a message body, by using the function:

```
INETGetURLex : fun [Chn S S S I fun [INET u0 S I] u1 u0] INET
```

The arguments are:

- the proprietary channel
- the verb (POST)
- the URL
- the message body
- a flag (leave at 0)
- a callback

- a user parameter

Response reception – and callback operation – are exactly the same as for the `INETGetURL` function.

You can interrupt a request in progress by calling the function:

```
INETStopURL : fun [INET] I
```

11. MULTIMEDIA PROGRAMMING

SCOL provides a number of multimedia possibilities in addition to 3D and the classic 2D interfaces.

You should note, however, that some of these possibilities are very much platform-dependent. Some APIs are not implemented on UNIX platforms: in this case the functions are present in their empty form (same type, but the function always returns `nil`).

Consult the reference manual for details of each multimedia API.

11.1. RealPlayer API

This API is used to read a RealPlayer flow (audio and/or video). The program simply defines the URL of the document to be read and is then notified of the state of progress. Sound is processed automatically and transmitted to the loudspeaker. Images are passed to the program in a bitmap, and the program then does what it likes with them: 2D display, use of bitmap as texture to apply the video to a 3D object, etc.

The API also has tools to control the flow (read, pause, position, volume, etc.).

In addition, the API is used to operate the login/password system defined by Real.

11.2. Quicktime API

This API applies the same principle as for RealPlayer. The functions are different, however, since both technologies are presented in a fundamentally different way as far as the developer is concerned.

11.3. Basic multimedia API

This uses multimedia players present on the system to play classic multimedia files such as Wav, Avi, Mpeg, etc.

In fact, the SCOL machine transmits files to the operating system, which will run them if it has the right drivers.

11.4. Audio API

The audio API has several components:

- mono recording/playback
- sample compression/decompression
- playback mixed with sound, possibly 3D

11.4.1. *Mono recording/playback*

This API is used to record sound by specifying the sampling frequency and sample size. It is also used to play back these samples by further specifying the frequency and size.

This API makes use of full-duplex possibilities if your hardware has them. If not, the playback start or record functions will return errors if you try to run both simultaneously.

11.4.2. *Audio compression/decompression*

This API is used for compressing and decompressing audio samples. The compression method is suitable for very low outputs since it goes down to a coding of 8 Kb/s, which represents a sixth of the pass band of a 57.6 Kb/s modem.

11.4.3. *Playback mixed with 3D sound*

This API (using *DirectSound* in *Windows*) is used to play back several samples simultaneously, while being able to specify a 3D position for each one.

11.5. Video API

The SCOL machine is able to use a video camera via this API and to capture images by specifying the desired capture frequency.

Users define a callback function that will regularly receive the images. They can then:

- convert them to bitmap in order to display or save these images
- compress them for possible transmittal.

The library provides extremely useful video compression/decompression functions for this purpose.

11.6. Printing

At the other end of multimedia are printing functionalities. SCOL enables two types of printing: raw text printing and bitmap printing.

For raw text printing, you will simply pass the character strings to be printed.

For bitmap printing, you will pass the bitmap and the print coordinates and size (with micron precision).

12. THE SCOL MACHINE: START-UP, CONTROL, STANDARD CLIENT AND SERVER

Up to now we have described the internal operation of the SCOL machine. Now we shall present a more global overview, encompassing the start-up and control of a SCOL machine. Moreover, a standard connection method has been developed to define the notion of browser, and we shall use this section to describe it in detail.

12.1. SCOL Voy@ger: the supervisor

When you start one of your SCOL programs, another SCOL machine always starts, characterized by the display of a window called SCOL Voy@ger. The SCOL Voy@ger allows the user to perform certain operations, but above all it acts as a **supervisor**, enabling the different SCOL machines present on the machine to be managed through a graphic user interface.

The supervisor automatically starts up when the first SCOL machine is started. Each SCOL machine is connected to the supervisor via a local TCP/IP socket, called **life-socket**: when this socket is closed by the supervisor the SCOL machine shuts down. Conversely, when this socket is closed by the SCOL machine, the supervisor is informed that the SCOL machine has shut down. This socket is also used by the supervisor to communicate with local SCOL machines, for example to command the console window to open or close.

The supervisor is a normal SCOL machine. What distinguishes it is that other SCOL machines automatically try to open a channel to it.

The precise use of the SCOL machine is as follows:

- **a. Started without argument :** If the supervisor is already present, nothing happens. Otherwise the SCOL machine starts in supervisor mode and then provides this function: see below for details.
- **b. Started with an argument :** If the supervisor is absent, another SCOL machine is started and sets itself up in supervisor mode.

Then the SCOL machine is started with the argument which gives the start-up script and possibly rights and memory size: see below for details.

12.2. Starting the supervisor

The supervisor presents an interface which provides the following functionalities:

- Starting SCOL machines whose start-up script is selected by the user from a drop-down menu.
- Maintaining a specific number of SCOL machines: these are started automatically when the supervisor starts up and are restarted when they are closed.
- Visualizing SCOL machines in operation, with start-up script name and time elapsed since start-up.
- Ability to destroy a particular SCOL machine and to display/hide the console window.
- Destruction of all the SCOL machines when the supervisor is closed.
-

The files that need to be present in the SCOL directory (usually ‘C:/Program Files/SCOL’) are:

- **usmress.ini**: text file containing the initial definition of the resource variables
- **usm.ini**: text file containing "hard" parameters:
  - `echo`: display mask
  - `port`: port used by the supervisor (1200 by default)
  - `log`: activation of log files
  - `logwin`: console display in the event of a runtime error. If adjusted to 'no', the machine stops immediately if there is an error (this is useful when you are working on a remote SCOL machine)
  - `forcedIP`: forces the definition of the local IP address (this is useful when a server does not know its apparent IP address from the Internet)
  - `HTTPproxy`: http proxy
  - `update`: SCOL version date
  - `scol`: name of the SCOL dll in use
  - `plugin`: name of a SCOL plug-in (this line may appear several times)
  - `disk`: SCOL partition (this line may appear several times)

12.3. Starting a SCOL machine with a start-up script

In this section, we describe the start-up command line of a SCOL machine.

The command line of the SCOL machine may contain up to three arguments (only the first is indispensable):

- the first gives the start-up script,
- the second gives the machine's rights (see below),
- the memory size to be assigned to the virtual machine.

12.3.1. Start-up script

The SCOL machine uses a start-up script file, which can be either a file or a character string. The script syntax was defined in the Channels and communications chapter.

On start-up, the machine creates an initial **unplugged** channel with a **minimal environment**. The script is then run in this channel.

The start-up script is on the face of it very short: loading one or more SCOL packages, then running a command.

The start-up script can be expressed in two ways:

12.3.1.1. Script file

In this mode you simply pass the script file name with the *.scol suffix. For example:

```
C:\Program Files\SCOL\usmwin.exe C:\Program Files\SCOL\Partition\test.scol
```

The name of this machine will be '`test.scol`'

12.3.1.2. Script string

In this mode, you directly pass the contents of the script, expressed in a rather special way:

- alphanumeric characters are retained

- spaces are replaced by '+' signs
- the other characters are replaced by a '%' followed by 2 hexadecimal figures

This "script" is preceded by the name of the machine surrounded by $ signs.

For example:
```
C:\Program Files\SCOL\usmwin.exe
$Tutorial/mytest.scol$%5fload+%22locked%2flib%2fconst%2epkg%22%0d%0a%5fload
+%22Tutorial%2fmytest%2epkg%22%0d%0amain%0d%0a
```

12.3.2. Operating rights

The second argument of the command line gives the machine's rights (these are rarely used). It is a character string such as: `CSDMRWK`

The SCOL machine has certain rights, each represented by a letter:

- `C`: client-networking (access to the `_openchannel` function with an address different from `nil`)
- `S`: server-networking (access to the `_setserver` function)
- `D`: distant-networking (access to the `_openchannel` function with a remote address)
- `M`: opening new machine without cache (starting a machine without its cache being activated)
- `R`: normal file reading (access in read mode to unsigned files)
- `W`: normal file writing (access in write mode to unsigned files)
- `K`: signed file writing (access in write mode to signed files)

The `C_Rights, S_Rights, ..., K_Rights` constants are defined in the SCOL language (they are whole masks). A SCOL machine can know its parameters:

`_getrights : fun [ ] I`
Returns the machine's rights (to be used with the previous masks).

`_setrights : fun [ I ] I`
Defines new rights in the direction of the restriction (necessarily below those of the machine).

12.3.3. Memory

The third and final argument of the command line gives the size of the memory used by the SCOL machine. This size is expressed in words of 32 bits. For example, `262144` will correspond to a memory of 1 Mb.

`_sizememory : fun [ ] I`
Returns the size of the SCOL machine's memory (counted in words of 32 bits).

`_freememory : fun [ ] I`
Returns the size of the SCOL machine's free memory (counted in words of 32 bits). This function triggers a Garbage Collector and is thus relatively slow.

12.4. Starting a machine or another process with a SCOL machine

One SCOL machine can start another by means of the following command:

_newmachine : fun [S_1 S_2 I_1 I_2] I
Creates a new machine whose name is S_1, with a start-up script S_2, and with rights I_1 and memory size I_2. Obviously, the new machine's rights are at most equal to those of the old machine.
If I_1 or I_2 are **nil**, the new machine inherits the old machine's values.

_newmachineS : fun [P I_1 I_2] I
Creates a new machine with the script file P, and with rights I_1 and memory size I_2. Obviously, the new machine's rights are at most equal to those of the old machine.
If I_1 or I_2 are **nil**, the new machine inherits the old machine's values.

_openbrowserhttp : fun [S] S
Historically, this function commands the standard browser on your machine to open by passing a URL as an argument. In fact, you can do more with this function; it depends on your URL's prefix:

- `http://` : opens your machine's standard browser
- `ftp://` : opens your machine's standard browser
- `mailto:` : opens your machine's standard browser
- `file://` : opens the file after the user has given his or her agreement through a dialog box.

Examples: there's no need to write a program to use the `_openbrowserhttp` function; use the SCOL Voy@ger and enter the following examples as URLs:

```
http://www.cryo-networks.com
mailto:scol.info@cryo-interactive.com
file://C:/Windows/calc.exe
```

12.5. Communication between the SCOL machine and the supervisor

The SCOL machine is linked to the supervisor by a normal channel. This channel is defined as being the **_masterchannel**, and the corresponding socket is called **socklife**.

The **_masterchannel** SCOL variable is accessible to users. They can make use of it to send messages to the supervisor.

Example:

```
defcom Copen=open S;;

...
_on _masterchannel Copen ["http://www.cryo-networks.com"];
...
```

The '`open`' message asks the supervisor to open a connection to a URL (http or SCOL).

The '`goto`' message does the same thing, then closes the sending SCOL machine.

The `socklife` socket has one distinguishing feature: when it is closed (shutdown or error), the SCOL machine stops.

12.6. Standard server and client

12.6.1. General remarks on SCOL machine communication

SCOL machines communicate by exchanging messages in the form of command+arguments (see the section on Channels and communications). When a message reaches its target, the latter searches to see if a function bears the same name as the command; if so, it executes this command.

In fact, when a SCOL machine sends a message to another SCOL machine, there is nothing on the face of it to guarantee that its correspondent has defined a function which bears the name of the command contained in its message. You can define a '**vocabulary**' notion: a SCOL machine's output vocabulary corresponds to the group of commands that it is capable of producing. A SCOL machine's input vocabulary corresponds to the group of commands that it is capable of interpreting, namely, the group of functions starting with a double underscore.

In order for two machines to communicate with each other, it is essential that the output vocabulary of one corresponds to the input vocabulary of the other, and vice versa. Otherwise, the messages sent by one machine will not be interpreted by the other: if Alice sends Bob a message '`f 1`', Bob has to have the `__f` function on his machine (more specifically, in the environment of the channel which connects him to Alice).

SCOL's distinguishing feature is that a machine's vocabulary is dynamic: you can at any time modify the environment of a channel, either by adding packages, or by subtracting packages. What this means is that you can modify the machine's input and output vocabulary at any time.

When Alice calls Bob's machine, Alice does not know then what vocabulary Bob is using, but as soon as Bob has told her which files to compile on her channel, Alice can modify her vocabulary and make it compatible with Bob's. Unfortunately, Bob cannot tell Alice which files to compile as he needs a common vocabulary to do this.

To solve this problem, we have defined what we call the standard server and client, which are in fact files written in the SCOL language. Their role is to initiate a communication between two SCOL machines. When Alice calls Bob, the following operations occur:

- Alice uses the standard client on the connection channel to Bob.
- When Bob receives Alice's connection request, he places the standard server on this channel.
  The standard server then gives the client the list of packages necessary to proceed further.
- The standard client checks that it possesses these packages.
- If there are some missing, it informs the server, which sends them to the client.
- When the requisite packages are in the client's possession, the server sends the client a start-up script.
- When the script has been sent, the server withdraws the standard server package and runs a script, called server script.
- Similarly, the client withdraws the standard client packages and runs the script that has been sent by the server, called client script.
- At this time, Alice's and Bob's vocabularies are fully synchronized (as long as Bob has sent the right packets). Bob's real server application can now begin.

12.6.2. Standard server - version 3

As we have just seen, the server that uses the standard server must supply several elements:

- It must initially place the standard server version 2 on the channels that connect it to its clients. This is done in the `_setserver` declaration. Example for a server on port 1285:
- `_setserver _envchannel _channel 1285 "_load \"locked/stdsrv3.pkg\"";`
- It must supply a server script. You need to declare the `scriptserver` variable:
- `var scriptserver="_load \"fs/fssrv2.pkg\"\n_contact";;`
- It must supply a client script. You need to declare the `scriptuser` variable:
- `var scriptuser="_load \"fs/fscli.pkg\"\nmain`
- It must supply the list of packages required by the client. You need to declare the `packsusers` variable, which is a list of type `[[S S S] r1]`: the first string is the name of a package, the other two must be empty strings.
- `var packsusers=["fs/fscli.pkg" "" ""]::nil;;`
- It must supply the minimal version number of the SCOL Voy@ger of the clients supported by the site.
- `var versionuser=0;;`
- This version number will be compared with the number returned by the `_version` function.

Example:

The client needs the file *"sample/foo.pkg"*. The client's script is `_load "sample/foo.pkg"\nstart`. The server's script is *_load "sample/bar.pkg"\nstart*. Then the server is written:

```
/* server */
var packsusers=["sample/foo.pkg" "" ""]::nil;;
var scriptserver="_load \"sample/bar.pkg\"\n_contact";;
var scriptuser="_load \"sample/foo.pkg\"\n_contact";;
var versionuser=0;;

fun main()=
_setserver _envchannel _channel 1285 "_load \"locked/stdsrv2.pkg\"";;
```

For those who are interested :

- the standard server uses the file: `locked/stdsvr3.pkg`
- the standard client uses the files: `locked/stduser.pkg`, `locked/stduser1.pkg`, `locked/stduser2.pkg` et `locked/IPrequest.pkg`

We will see in the 'Integration in a Web page' section how to start the standard client "manually", but in fact the SCOL Voy@ger takes care of it for you when you enter a URL without a prefix or starting with `scol://`

Note : Versions 1 and 2 of the standard server exist; their file names are "`locked/stdsvr.pkg`" and "`locked/stdsvr2.pkg`". Version 3 is similar to version 2, but uses the new '`zip`' compression instead of '`mzip`'.

13. INTEGRATION POSSIBILITIES

SCOL offers many gateways to other technologies:

13.1. Interfacing via file exchange

This is of course the most obvious method, which is used mainly on the server side: since the SCOL machine has access to all the files contained in the SCOL partitions, all that is required is for the technology that you want to interface with SCOL to produce or use files in these partitions, and for your SCOL server to produce or use the same files.

13.2. Interfacing via database: SQL library

This is used only on the server side: the server accesses data from another information system via SQL databases (using Odbc technology)

13.3. Interfacing via http: server or client http libraries

On the SCOL server side, the two libraries are used indiscriminately:

- **http client**: the technology that you want to interface with SCOL is a web technology (cgi, asp, etc). Consequently, your SCOL server can itself become a client of this web technology by performing http requests itself thanks to the client http library.
- **http server**: the technology that you want to interface with SCOL is a web client technology (often used in payment systems for example: the payment server makes an http request either itself or via the surfer's web browser, indicating the result of the operation: success or failure). Your SCOL server can thus receive and process these requests by opening an http server thanks to the server http library.

On the SCOL client side, the client http library will mainly be used. This allows the SCOL client to access any given web services, in other words to make a request and analyze the reply. For example, to represent in 3D the result of a search on an Voy@ger such as Altavista or Yahoo, the SCOL client will make the search request, receive the reply, analyze it and convert it into a 3D representation.

13.4. Integration in a web page

13.4.1. Simple interfacing

SCOL can be run as a Netscape plug-in or as an ActiveX component for MSIE or visual Basic.

To integrate SCOL in a Web page, simply enter the following code (the example here describes a 500 by 400 point zone):

```
<body onLoad="Load()" bgcolor="#FFFFFF" text="#000000" link="#00CCCC">
<object
id="scol"
name="scol"
classid="clsid:7A96FF35-4937-11D1-8F2C-00609779BDA3"
codebase="http://www.cryo-networks.com/files/atlscol.dll"
align="middle"
```

```
border="0"
width="500"
height="400">
<embed
align="baseline"
border="0"
width="500"
height="400"
name="scol"
pluginurl="http://www.cryo-networks.com/files/scp10.exe"
pluginspage="http://www.cryo-networks.com/files/scp10.exe"
type="application/x-scol"></embed>
</object>

<script LANGUAGE="JavaScript">
<!--
function Load()
{
  if(navigator.appName=='Netscape')
  {
document.scol.LaunchMachine('$browser$%5fload+%22locked%2fstduser%2epkg%22%0amain+%22scol%2ecryopolis%2ecom%3aCryopolis%22+ffffffff+NIL CSDMRWK 262144',1,0);
  }
  else
  {
scol.LaunchMachine('$browser$%5fload+%22locked%2fstduser%2epkg%22%0amain+%22scol%2ecryopolis%2ecom%3aCryopolis%22+ffffffff+NIL CSDMRWK 262144',1,0);
  }
}
//-->
</script>
```

There are in fact differences between Netscape and MSIE in the way SCOL is started, but in both cases you use a `LaunchMachine` function, by passing the command line as an argument, here a "standard user" client, to port 220.87.26.15:3005

You will notice the syntax of the `Launchmachine` functions' argument. It consists of a character string containing three words separated by spaces:

- the machine start-up script: `'$name$script'` type, the script being in " `strtoweb` " format.
- the machine's rights (generally CSDMRWK)
- the size of the memory given to the machine.

In general there will be two types of script:

- the script starting the SCOL machine in a web page, which is the " stduser " script:

```
_load "locked/stduser.pkg"
main "url" ffffffff NIL
```

Which, by applying the strtoweb function (replaces spaces with + and all non-alphanumeric characters with %ascii_code_in_hexadecimal) for the `scol.cryopolis.com:Cryopolis` url, gives:
`%5fload+%22locked%2fstduser%2epkg%22%0amain+%22scol%2ecryopolis%2ecom%3aCryopolis%22+ffffffff+NIL`

However, since the SCOL machine applies the webtostr function on this string, a function which only looks for the characters + and % and leaves the others unchanged, it can be written more clearly:

```
_load+"locked/stduser.pkg"%0amain+"scol.cryopolis.com:Cryopolis"+fffffff+N
IL
```

Which gives:

```
...LaunchMachine('$browser$_load+"locked/stduser.pkg"%0amain+"scol.cryopoli
s.com:Cryopolis"+ffffffff+NIL CSDMRWK 262144'...
```

N.B. the quote mark (") character can only be used provided the apostrophe (') is used as a delimiter of the character strings in Visual Basic or Java Script : ...LaunchMachine(**'**$browser...)

- the script starting the SCOL machine in autonomous mode (the Web page starts a SCOL machine that is not integrated in the page, but that is not destroyed when you change pages or close the browser):

```
_load "locked/link.pkg"
main "url"
```

This will produce the following script string:

```
_load+"locked/link.pkg"%0amain+"scol.cryopolis.com:Cryopolis"
```

Which gives:

```
...LaunchMachine('$browser$_load+"locked/link.pkg"%0amain+"scol.cryopolis.c
om:Cryopolis" CSDMRWK 262144'...
```

13.4.1.1. passing of parameters in the url

If your site uses DMS architecture (site created with SCS), the url can also be used to pass parameters to machines (this involves resource variables that the SCOL machine will be able to read thanks to the `_getress` function).

The form of the url is then:
`Name_machine:port_or_name_service/resource1+value1/resource2+value2…`

For example, you want to pass the resource variable 'login' with the value Alice, and a resource variable 'authorization_number' with the value 123456, for the Cryopolis site. The associated url is:
`scol.cryopolis.com:Cryopolis/login+Alice/authorization_number+123456`

The SCOL machine will be able to read the values 'Alice' and '123456', naming them `_getress "login"` and `_getress "authorization_number"` respectively.

Comment: if the value contains special characters, such as `+` or `'`, you should remember that your string will be read in " `webtostr` " format and therefore replace these characters with `%ascii_code_hexadecimal`.

Furthermore, if you have used characters such as spaces, carriage returns, etc., bear the following in mind: the program assumes that between two " `/` " characters in the url, it can apply the `webtostr` SCOL functions then `strextr` to obtain a list (`resource: :value: :nil) : :nil`

If your site does not use DMS architecture (which should be quite rare), you recover all these resource definitions in the form of a list `[[S r1] r1]` containing one line per definition, in the " parameters " variable.

Instead of `_getress "login"`, you will be able to use `switchstr (strextr _getress "parameters") "login"`

13.4.1.2. using the window supplied to the component by the container

The SCOL machine started as above as an **ActiveX** component can use the zone which the container allocates to it by using the following function:

`_GETactiveXWindow : fun [Chn I S] ObjWin`

The integer is the window's flag, the string is the window's name. This function returns `nil` if the window is not available, which is the case if:

- the SCOL machine has not been started as an ActiveX component or a Netscape plug-in
- the component's zone has been defined in an invisible web page (only for certain browsers)

13.4.2. Sophisticated interfacing

In some applications it is useful to have a Web page communicate with the SCOL component that it contains by using JavaScript in the Web page. This is perfectly possible with SCOL.

The starting of the component is different in Netscape, because to make this communication possible the component must be started by Java (and not *JavaScript*). The basic Web page is therefore the following:

```
<body bgcolor="#000000" text="#7DcDe7" link="#FFFF00"
vlink="#FFFF00" alink="#FFFF00" onLoad="Load()">
<OBJECT ID="scol"
        NAME="scol"
        WIDTH=500
        HEIGHT=400
        align="baseline"
        border="0"
        CLASSID="CLSID:7A96FF35-4937-11D1-8F2C-00609779BDA3">
<EMBED name=scol
       type=application/x-scol
       border="0"
       width=500
       height=400>
</EMBED>
<SCRIPT LANGUAGE="JavaScript">
<!--
function Message(txt)
{
  selectmsg(txt);
}
//--></SCRIPT>
<applet name="myscol" code="SCOLTest.class"
            width=5
        height=5
            mayscript>
</applet>
</OBJECT>

<script LANGUAGE="JavaScript">
<!--
function Load()
{
  if(navigator.appName=='Netscape')
  {
```

```
document.myscol.run('$browser$_load+"locked/stduser.pkg"%0amain+"scol.cryop
olis.com:Cryopolis"+ffffffff+NIL CSDMRWK 262144',document);
  }
  else
  {
scol.LaunchMachine('$browser$_load+"locked/stduser.pkg"%0amain+"scol.cryopo
lis.com:Cryopolis"+ffffffff+NIL CSDMRWK 262144',1,0);
  }
}
//-->
</script>

<script language="JavaScript"><!--
function selectmsg(msg)
{
...
}
//--></script>

<script language="VBScript"><!--
Sub scol_Message(msg)
  call selectmsg(msg)
end sub
--></script>
```

In this example, the selectmsg function receives the messages sent by the SCOL machine. These messages are sent by the function :

`_onX : fun [Comm] I`

N.B.: For reasons known only to Netscape's developers, the JavaScript code processing the message must specify complete URLs rather than relative ones, otherwise the Java machine crashes without warning.

To send a message to the SCOL machine, the latter must first define the channel supposed to receive them: for this definition you use the function:

`_setX : fun [Chn] Chn`

Then you just write in JavaScript, for example:

```
if(navigator.appName=='Netscape')
{
 document.scol.SendMessage('_f 1 "abc"');
}
else
{
 scol.SendMessage('_f 1 "abc"');
}
```

This message will be received by the channel defined by the `_setX` function. The SCOL machine will then search for a _f function taking two arguments, an integer and a string. The message's format is that of SCOL messages and therefore that of the scripts.

13.4.2.1. ActiveX integration in container mode

ActiveX technology is a Microsoft technology, available only with Windows. It is based on the concept of components and container:

- a component is any given functionality that is relatively autonomous and which generally requires a window in which it can offer a graphic interface, and which can communicate externally via an incoming and outgoing API (sending/receiving of messages, which are in fact sets of calls to functions and callbacks)
- a container is a document (typically a window), in which you define a certain number of zones (sub-windows) in which you activate ActiveX components. Once the components have been activated, the container can communicate with them via the component's API.

SCOL technology is both an ActiveX component and container. The ActiveX component mode is used to integrate SCOL in the Internet Explorer browser (which is itself an ActiveX container). The ActiveX container mode can be used on both the client side and the server side:

On the client side, it is used to integrate - in the SCOL client interface - functionalities available in the form of ActiveX components. For example, a Web page (Internet Explorer is also an ActiveX component), a particular viewer, etc. However this presents two limitations (which only exist if your service is open to the public):

- Since ActiveX technology is only available on Windows, only visitors to your site who use Windows will be able to access these functionalities
- SCOL does not handle, for security reasons, the downloading of your ActiveX components. You therefore have to perform this operation yourself. However, the SCOL program can detect the presence of an ActiveX component.

13.4.2.2. Interfacing via BSD socket (Telnet library)

The SCOL machine has a Telnet API (client BSD sockets). This can be useful on the server or the client side:

- on the server side, you can open a connection to any given TCP/IP service.
- on the client side, you can also connect to any given TCP/IP service. However, most firewalls block this type of connection, and there is no way of getting round it. It should be noted however that the SCOL Telnet library is compatible with SocksHost type proxies.

13.4.2.3. Interfacing via SCOL socket

It is always possible to connect to a SCOL machine by passing yourself off as another SCOL machine. This makes interfacing far simpler, as it takes advantage of the parsing and channel management that are already integrated in a SCOL machine. Traditional TCP/IP sockets are used for this with the following protocol:

each message consists of two parts:

- a two-byte header coding the size of the message body (in the order low byte, high byte)
- a message body which is in fact a line of SCOL script (refer to the relevant chapter for details about the syntax).

14. DMS PROGRAMMING: DISTRIBUTED MODULES SYSTEM

This chapter describes the architecture of programs referred to as DMS. This is a "component" type architecture; as such, its purpose is to save developers from having to redevelop everything with each new application, by allowing them to use components from other applications. For this reason, we cannot give a complete example of its use. We will content ourselves with presenting some examples of components.

14.1. Presentation

SCOL technology is based on a programming language that integrates Internet communication possibilities associated with a certain number of graphic, 3D, multimedia, SQL, etc. libraries. The first objective of the SCOL technology is therefore reached: to provide developers with a simple and powerful tool that makes development faster and more reliable. The technical difficulties are resolved by the technology; consequently developers need only concentrate on the real problems, those specific to the application they are developing.

The technology's second objective is more ambitious: SCOL technology can also be of interest to creative users who are not necessarily experienced in programming techniques, in other words, users who would adopt an integrator approach, taking a graphic element here, an interfacing element or part of a program there, to subsequently build a distributed-type application, a virtual world for example.

For this purpose alone, it would have been possible to program a kind of Wizard that would ask users for their preferences, suggest some options and build a ready-to-use application. This approach would soon have proved limited and disappointing, since such a tool would offer no flexibility.

The solution chosen was to define a particular method of using SCOL technology. This method makes it possible to homogenize programming developments by introducing the concept of the "module". Since the modules are distributed (in the IT sense of the word), the programming method is known as DMS: Distributed Modules System.

A module is part of a program that performs a given function. Seen from the outside, modules all resemble one another, a bit like an integrated circuit: the case is the same, the pins are all similar, only the number, the direction (incoming or outgoing) and the function of the pins are different. To create an application, all you need to do is assemble the modules and create links between the pins. This is done **with the mouse and without programming.**

The modules are distributed: one part runs on a machine called the 'Server', the other part runs on user machines called 'Clients'. Deciding what should be calculated on the server and what should be calculated on the clients is a complex IT problem, which is out of the general public's reach. The module therefore removes this problem: it is not up to the person who assembles these modules to determine the question, but to a module's developer.

There are other systems based on modular programming. The original aspect of DMS is twofold:

- Modules are assembled by creating links between the modules, and not using a programming language, often referred to as a script language. Having to use such a language, even if simple, puts the use of these architectures outside the reach of the general public

- The modules are distributed, but the user does not need to concern him or herself with it. He or she does not need to define one assembly of modules for the server and another for the clients; only one assembly needs to be defined.

DMS architecture is not only useful for the general public, it also saves the developer a lot of time. Developing a DMS module is easy. Transforming a simple function into a DMS module means you do not have to develop functionalities already present in other modules: log files, console windows, passwords, statistics, etc. In this way the DMS constantly evolves, for everyone's benefit, as new modules are developed.

To develop a DMS module, two or three program parts need to be written:

- **the server module** : program running on the server
- **the possible client module**: if part of the processing is carried out on client machines (mainly interfaces), you must write the program that will run on the clients
- **the module editor**: this editor will integrate itself in the editor of the DMS sites (SCS)

Each of these parts of programs uses an API the details of which are given further on.

14.2. Definitions

Let us begin with a few definitions.

Dms Site
Distributed application complying with the architecture described in this document

Client
Software running on the computer of a Dms site user

Server
Software putting users into contact. There is one server for each Dms site.

Module
Constituent element of a Dms site. Usually distributed, it has a server part which runs on the server side, and a client part which is duplicated on the client machines. To make discussion simpler we will say 'server module' for 'server part of a module', and 'client module' for 'client part of a module'.
Similarly, we will say 'associated client module and server module' for 'client and server parts of a module'.

User
Generalizing concept of a user present in the site. This user is either real (in which case it corresponds to a client), or virtual (meaning an entity stored on the server). The users move along the links.
Exactly one user corresponds to each client.

Event
Signal leaving a module. The event is either a client event (the source of the event is a client), or a server event (the event is produced on the server): this localization is decided by the module's author. A User is generally associated with such a signal: in fact the signal is the sign that the User is "moving".

Action
Signal entering a module. The action is either a client action (the action is usually processed by a client module), or a server action (the action is processed by the server module): This localization is decided by the module's author.

Message
Message sent between a server module and an associated client module (in either direction).

Link
Association of an event of one module with an action of another module, possibly subject to conditions. You can assign a parameter to a link. This will be referred to as the 'link's parameter'.

Zone
Rectangular graphic zone used by a module to present a result, an interface, etc.

Document
Graphic window in which one or several zones can be defined. The documents are organized in a hierarchy. This hierarchy defines two types of child windows: " popup " child documents (opening above the parent document), and simple child documents, corresponding to a zone of the parent document. A main document is one which doesn't have a parent document.
A Dms site uses two main documents, the server document and the client document.

14.3. Principles

14.3.1. Module architecture

The architecture of a Dms site is modular: a site is made up of a variable number of modules, connected to each other via links. Each module manages one or more functionalities: log, authentication, 3D space, etc. Each module can have a distributed operation: one part of the processing is performed on the server, another on the clients.

The role of a Dms site's author is therefore to select a certain number of modules and assemble them by creating links between them.

One of the major problems in creating an application which puts different users into contact is to establish how the processing will be divided between the server and the clients. This problem is eminently technical, and therefore beyond, a priori, someone who would be building a Dms site. The problem of distribution will therefore be resolved within a module, by whoever has developed the module.

Generally, each module will be divided in two, one part running on the server, another on the clients. The communication between the client part and the server part of a module will be managed exclusively by the author of the module, using the traditional communication techniques included in SCOL.

Communication between the modules will be in the form of links, and will therefore be defined by the site's author.

The modules each have a name, the only restrictions on which are that:

- two sibling modules must have different names (see **encapsulation** later on to know more about sibling modules)
- a name must not start with the character '.'

14.3.2. Tree of documents

A module can at any time request the use of one of a document's zones, or on the contrary, stop using a zone. The system manages the display of the documents in such a way that the document is

visible as soon as at least one of its zones is currently being used by a module. A document that no longer has a zone in use is destroyed, except for the main document, which is the parent of all the others, and shows the application's presence.

14.3.3. Encapsulation

The modules can be encapsulated: a set of modules can be grouped together and replaced by a black box which contains them. The modules are therefore organized in a tree of modules whose non-leaf nodes are black boxes. Each module can then be linked to a sibling module, a child module or the parent module. The black box acts as a transparent relay: actions are directly connected to events, in both directions.

14.3.4. Inter-module links and communication

A link joins one of a module's events to one of another module's actions. The number of links attached to a given event or a given action is not limited. The links are directional: from the event to the action. A default parameter and/or a condition is associated with each link.

A module can trigger an event at any time. The system converts it, according to the links, into actions for other modules. Each module concerned by the action is informed of the sender's identity, and can reply, if the sender has provided for this, using a system of " tags ".

Two types of inter-module communication can therefore be defined:

- **communication by an event**: un event is sent to the system, which converts it into actions, in compliance with the links laid out by the site's author. The event can be issued by the server module or the client module. The action can be received by the server module or the client module. However, the client->server link can present a security weakness: there is nothing to prove that the client module is running correctly.
- **communication by a reply**: when a module issues an event, it can associate it with a reply " tag ", which is actually a callback function that expects to receive a parameter and/or a list of Users. The module that receives an action associated with this event can then reply to this " tag " by providing a parameter and/or a list of Users.

In fact an event represents the " movement " of a User, leaving a module via a particular pin. Links leading from this pin take the User to other modules. This doesn't mean however that the User quits the module that has produced the event: a User can be ubiquitous, meaning he or she can be present in several modules at the same time.

For this reason, an event is defined by the following elements:

- **a User**: someone who moves around in the module graph
- **a parameter**: a given character string. If the value of this string is nil, the parameter is replaced by the first non-nil default parameter associated with the links that the event follows
- **a list of Users** connected with the event
- a reply " **tag** "

Each of these elements is optional and can be replaced by `nil`. However, it is extremely rare for the User argument to be `nil`, since an event is in fact the movement of a User.

If an event is produced by a client module, the associated User is implicitly the one that corresponds to the client module.

If an event leads to an action that is usually located on a client module, the routing carried out depends on the associated User:

- if the User is real (i.e. corresponds to a client), the action is processed by the module of the associated client
- if the User is virtual, the action is processed by the server module

This means that an event's User parameter allows all routing problems to be resolved.

14.3.5. Dynamic activation

Creating a site consists in assembling a set of modules. Each machine (client or server) has an approximate copy of this site: this means that certain modules are not represented on each machine. For example, only the 3D space module where a user is located is represented on this user's machine; however, all the site's 3D space modules are represented on the server. This observation brings up the question of the dynamic activation of the modules: while the modules are all active on the server side, only a few modules are active on the client side: the number and nature of the active client modules varies according to the time and the client.

In fact, it is the server module that triggers the activation of the client module on a given client machine. For each server module, the system manages the list of activated client modules, which allows it in particular to filter messages, thus guaranteeing security in the exchange of messages. Activation is not therefore automatic: the fact that an event is linked to one of a module's actions does not mean that that this module will be created if the event occurs. If the event occurs when the module receiving the action has not been created, the event is simply ignored.

To activate a module, a certain amount of data must be transmitted to the client, and the client must, if necessary, download a certain number of files (in particular SCOL source files describing the operating of the client module). Activating a module on a client is therefore a complex operation that may take some time. During this time, the client module is “ dormant ” and buffers the messages and actions it receives until it can be properly started: once all the necessary files are present, the client module is compiled, started, and all the buffered messages are processed.

14.3.6. Users and UserInstances

We have seen that Users are the mobile elements of the DMS architecture: they move along the module graph when events occur. A module therefore receives a flow of Users.

When the module considers it useful, it can define a structure known as a `UserInstance`. For a given module, only one `UserInstance` can be defined per User. The `UserInstance` object is an object:

- associated with a User,
- distributed and synchronized between the client and server parts of a module, and offering highly practical possibilities for communication between these different parts.

14.3.7. The SCS site editor

The site editor (whose commercial name is SCS for) is the tool linked with the DMS architecture. It is used to:

- select modules to be integrated in the site: create, remove
- define server and client documents
- define links between modules
- assign zones to the modules: indicate in which zone the 3D will be displayed, a button, an image, etc.

- start the module editors

Indeed, each module usually contains an editor that is used to define its parameters. For example:

- defining a 3D space for a module managing 3D space
- defining the texts of a display banner module

A distinction will be made between the two types of editor: site editors and module editors. The module editor is usually written by the module developer.

14.4. Downloading of resources

The server also plays the role of resource distributor. A resource is a given set of bytes. A resource is named and belongs to the module that is its owner. Resources are the data that the clients will be able to download: files containing the client module's program, graphic, 3D or sound files, etc. It is important that each module records the resources that the client may need, as this ensures automatic update and mobile code possibilities: otherwise only clients that already have these files will be able to operate normally. This may be an oversight or a deliberate omission to select clients. When a module is being developed, it is important to check that when a "first time" client connects to the site, it will indeed download the client data specific to the module.

A resource is usually accessible to any client whose client module corresponding to the proprietary server module has been activated. However, the resource can be protected by limiting access only to authorized clients.

On the server side, the resource may be present in the server's memory or stored on disk. Only in the first case can it be compressed before being downloaded by the client. In the second case, the data will be directly transferred: it is therefore recommended that the file already be compressed (jpeg graphic file for example).

On the client side, the resource can be stored in the cache under a given name that would show – when subsequently used - whether the resource needs to be downloaded or if it is already present. The resource can also be provided directly to the client, without saving it to the disk first.

The server calculates the signature of the resources submitted to it. This signature is transmitted to the client module when it is created. The client module checks the signature of the file bearing the name of the resource and, if it doesn't correspond, downloads the resource again. Checking that a client does indeed have a given file is done by comparing the signature of its content, and not by looking at its creation date, as this latter method is much less reliable.

A server module can:

- record a resource by specifying its name
- unrecord a resource by specifying its name
- unrecord all the resources
- authorize a client to access a resource

A client module can:

- request the downloading of a resource if it knows its name, and choose whether or not to go via the disk cache
- stop the downloading of a resource
- stop all the downloads that it has requested.

A client can also transfer a resource to the server, through an upload mechanism.

N.B.: when you are developing a module and are working on the client part, remember to restart the server in order for the changes made to the client code to be accepted. Indeed, the server calculates

the signature of the resources when it is started. If you change a resource without restarting the server, the signature of the old resource stays in its memory. If you then start a client, the server will transmit to it the old resource, which will be placed in the cache partition. This will mask the new resource, and you will be left wondering why your changes have not been accepted.

14.5. DMS site definition files

DMS sites are defined through two types of files.

Dmc (Distributed Modules Class) files define a module class: for example, a log management module, a 3D management module, a text interface module, etc. A Dmc file contains the list of files used by the module class -typically parts of programs - which will be automatically recorded as resources on the server, then the module start-up scripts, one for the server, one for the client, and one for the editor.

The Dms file contains the definition of a site:

- the module graph (tree of modules and inter-module links)
- the definition of the client and server documents
- the parameters defined for each module

It is worth specifying the different elements' origins:

- the Dmc file is created by the module developer
- the Dms file is created by the site editor (SCS)
- the definition blocks of a module (starting with the 'dmi' block) are created by the module editor

14.5.1. DMC file: distributed modules class

A dmc file describes a module class.

- `name [name of class]`: name of the class (for the editor)
- `register [file 1] ... [file n]`: list of files to be saved by loading them into the memory
- `registerF [file 1] ... [file n]`: list of files to be saved without loading them into the memory
- `serverNeeded [file 1] ... [file n]`: list of files required to start the server
- `serverLoad [file 1] ... [file n]`: list of files to be successively compiled to start the server
- `clientNeeded [file 1] ... [file n]`: list of files required to start the client
- `clientLoad [file 1] ... [file n]`: list of files to be successively compiled to start the client
- `editorNeeded [file 1] ... [file n]`: list of files required to start the module editor
- `editorLoad [file 1] ... [file n]`: list of files to be successively compiled to start the editor
- `bitmap [file]`: name of the file containing the default icon to be used by the site editor.
- `helpFile [ help file`: help file for the editor

All the file names are:

- Either absolute: for example `'dms/admin/log/log.pkg'`

- Or relative: descending in relation to the dmc file directory. This means that they start with an '`./`': for example '`./log.pkg`'. The sequence '`../`' is not recognized.

The files for the '`…Load`' lines do not need to be rewritten in the '`…Needed`' lines: the system assumes that they must be necessary for the module to start (since they have to be compiled).

To determine the list of files that are useful to the site (with a view to duplicating the site on another server for example), the system concatenates the lists `register`, `registerF`, `serverNeeded`, `serverLoad`, `clientNeeded` and `clientLoad`.

Let us now describe the start-up mechanism of a server module. When a server module is created, the system checks for the presence of the '`serverNeeded`' files. It then creates an unplugged channel that inherits the server API, successively compiles the files for the `serverLoad` line in this channel, then runs the `IniDMI` function (see further on).

Now let's turn our attention to the start-up mechanism of a client module. When the module is activated on a client machine, this machine starts by checking that it has the Dmc file. If this isn't the case, it downloads it from the server. This file does not have to be declared in the list of resources ('`register`" and '`registerF`' lines), as this is done automatically for all the modules that are present on the server.

Then, the client makes sure that it has all the files for the '`clientNeeded`' line. If any are missing, it asks the server to supply them. This is only possible if these files appear on the '`register`' or '`registerF`' line. If only one file is missing, a dialog box appears on the client, which is subsequently interrupted.

When all the required files are present, the client module is created as follows: an unplugged channel is created, inheriting the client API, then the files for the `clientLoad` line are successively compiled, then the `IniDMI` function (see further on) is called.

The files placed on the '`register`' line are processed as follows:

- they are read, then compressed (**zip** function)
- then they are kept in the SCOL machine's memory
- they are transmitted to the client that requests them

The files placed on the '`registerF`' line are processed differently:

- they are read, but not compressed
- they are not kept in the memory, but stored on disk
- when a client asks for them, they are reread on the server's disk

The only advantage of '`registerF`' files is that they don't take up too much of the server's memory. This is why the 3D module puts textures on this line. The fact that there is no compression is not a drawback since most textures are in jpeg format, which is already compressed to a considerable degree.

As regards the editor, to start the module editor, the site editor checks that the 'editorNeeded' files are present. If only one is missing, a dialog box appears to indicate that the editor cannot be started. If all the files are present, the editor creates an unplugged channel that inherits the editor API, successively complies the '`editorLoad`' line files, then calls the '`IniEditor`' function (see further on).

In the previous version, there were the lines `serverScript`, `clientScript` and `editorScript`, which defined `the start-up script` for the server, client and editor parts. These scripts were always a series of `_load`. Consequently, this system has been replaced by the lines `serverLoad`,

clientLoad and editorLoad, which are easier to use and offer the possibility of defining relative paths. However, upwards compatibility is guaranteed.

14.5.2. Dms files

The Dms file represents a module tree. Each node contains a number of named definition blocks, and no two blocks can have the same name. A definition block consists of a list of 'name {value1 {value2 … {valueN}…}}' lines. The values are strings of bytes.

The site's main node is in fact the "black box" that contains the whole of the site. It contains the site's global definitions (name, port, documents, etc.).

The syntax of the dms file is as follows (the file is in strextr format):

```
Module : :=
module name server_number
(Definition)*
(Module)*
endmod

Definition : :=
def name
(Line)*
enddef

Line : :=
> name (value)*
```

Each leaf module contains:

- A **'dmi'** definition block containing:
  - the class (in other words the name of the dmc file)
  - the events
  - the actions
  - the zones
  - the list of files it uses and which are therefore part of the site (images, textures, 3D, sound, etc), with a view to easily "duplicating" a site from one server to another
- A **link** definition block containing:
  - the module's outgoing links
- A definition block **'zone'** containing:
  - the correspondences between zones and documents

Each non-leaf module contains:

- A **'dmi'** definition block containing:
  - the incoming pins ('in')
  - the outgoing pins ('out')
  -
- A **'links'** definition block
  - the links leaving one of the pins

The global module (main node) that contains:

- A **'def'** definition block containing:
  - the site's global definitions:
    - name
    - port

- `timeout`

- A '**docserver**' definition block containing:
  - the description of the server document
- A '**docclient**' definition block containing:
  - the description of the client document (formerly Scc file)

Now let us detail the fields of each definition block type.

14.5.2.1. 'Dmi' definition block: distributed module instance

A **dmi** definition block describes the basic parameters of a module instance, or a black box.

In the case of a module instance we find:

- `name [instance name]` : name of the instance
- `class [dmc file]` : class file
- `event [name]` : event produced on the server
- `eventC [name]`: event produced on the client
- `action [name]` : action on the server
- `actionC [name]` : action usually on the client
- `zone [module zone name]` : alias of a zone used by the module (for the editor)
- `zoneC [module zone name]` : alias of a zone used by the client module (for the editor)
- `register [file 1] ... [file n]`: list of the files to be saved by loading them into the memory
- `registerF [file 1] ... [file n]` : list of the files to be saved without loading them into the memory
- `serverNeeded [file 1] ... [file n]` : list of the files that are useful to the server, with a view to duplicating the site on another machine.
- `bitmap [bitmap file]` : bitmap to be used in the editor

Here the file names are absolute.

The fields defined above are standard fields. Some modules will define additional fields that are specific to them: advertisement texts for a display banner, address of a directory for an automatic recording module. However, a more elegant solution would be to put this specific data in different definition blocks.

Events and actions are defined either on the server or on the client, which is why there are the lines `event/eventC`, `action/actionC`. Determining whether an event (or an action respectively) needs to be defined in the `event` or the `eventC` line (or the `action` or `actionC` line) is very simple: everything depends on the module that causes the event (resp. that processes the action), server or client.

The `register` and `registerF` lines must contain the files specific to the instance. Here it is pointless to redeclare the files contained in the Dmc. The same comments will be made on the '`register`' and '`registerF`' files as for the Dmc files.

The `serverNeeded` line is for determining the files that are useful to the site and are not already in the `register` and `registerF` lines.

For a black box, the following fields will appear in the dmi definition block:

- `name [name]` : name of the black box

- `in [name]` : incoming pin
- `out [name]` : outgoing pin

14.5.2.2. 'Link' definition block

The 'link' definition block only contains lines in the following form:

`[event] [destination] [action] [param] [reply] [condition]` : description of a link assigned to a module event The destination module is defined by the 'destination' field as follows:

- `..` : parent
- `.name` : child
- `name` : sibling

The `reply` parameter here is for upwards compatibility.

The conditions for links are a string intended to be used by the `strextr` function. Each line corresponds to an activation condition: there is a logical OR between the lines.

Each line is made up of a list of basic conditions. Each condition corresponds to one or more words in the line. All the basic conditions of a line must be filled in: there is a logical AND between each basic condition.

The first word of a basic condition shows what type of condition it is, the words that follow give the arguments. There are nine conditions:

- `!` : reverse condition of the rest of the line
- `login`: does the login have a particular value (1 parameter)?
- `notlogin`: is the login different from a particular value (1 parameter)?
- `ip`: does the IP address have a particular value (1 parameter)?
- `notip`: is the IP address different from a particular value (1 parameter)?
- `item`: does the user have a particular object (the parameter is the object's reference)?
- `noitem`: does the user not have a particular object (the parameter is the object's reference)?
- `items`: does the user have a particular object in a particular quantity (2 parameters: object reference and quantity)?
- `items`: does the user not have a particular object in a particular quantity (2 parameters: object reference and quantity)?
- `ActiveX`: the client uses SCOL in ActiveX component mode (0 parameter). In this mode the browser should not usually be started from SCOL, as this would change the current page (the one that SCOL is running in), and would therefore destroy the client. It is better for the Web page to be programmed (*javascript/vbscript*) to open a new frame.

14.5.2.3. 'Zone' definition block

This is made up of lines in the following form:

`zoneS [dmi name] [tree zone name]` : correspondence between the name of a zone defined in the dmi file and a zone defined in the scs file document tree

`zoneC [dmi name] [tree zone name]`: the same for a zone used by the client module

14.5.2.4. 'Def' definition block

The following fields are found:

- `name [name]` : name of the site
- `port [number]` : server port number
- `timeout [time in seconds]` : maximum client response time to a signal

The timeout is used to detect clients who have disconnected themselves accidentally. The TCP/IP protocol means the timeout may last several minutes before the server detects the disappearance of a client. Here, a signal is sent at regular intervals to all the clients. The clients simply need to reply to this signal with another signal. When the server sends a signal, it checks that it has received the previous signal, otherwise it disconnects the client itself.

14.5.2.5. 'docclient' and 'docserver' definition blocks

These definition blocks have the same syntax and define the client and server documents respectively.

`doc [name] [type] [resizeFlag] [x1] [y1] [x2] [y2] [w] [h] [bitmap] [bitmapFlag] [color]` : definition of a document, possibly with a document background image.

`zone [name] ] [resizeFlag] [x1] [y1] [x2 [x1] [y1] [x2] [y2] [w] [h] [color]` : definition of a zone of the document previously defined

`... recursion`
`enddoc` : end of the document definition (compulsory)

The values for document [type] are:

- `0` : indicates that the document is an internal document
- `DOCpopup` (1): indicates that the document is popup in relation to its parent document. Otherwise, the document's name must be that of a zone of the parent document.

The values for document [resizeFlag] are :

- `ZONE_LW_FLEX` (1): the left border between the zone and its father can be changed
- `ZONE_MW_FLEX` (2): the width of the zone can be changed
- `ZONE_RW_FLEX` (4): the right border between the zone and its father can be changed
- `ZONE_LH_FLEX` (8): the top border between the zone and its father can be changed
- `ZONE_MH_FLEX` (16): the height of the zone can be changed
- `ZONE_RH_FLEX` (32): the bottom border between the zone and its father can be changed

The values for document [bitmapFlag] are :

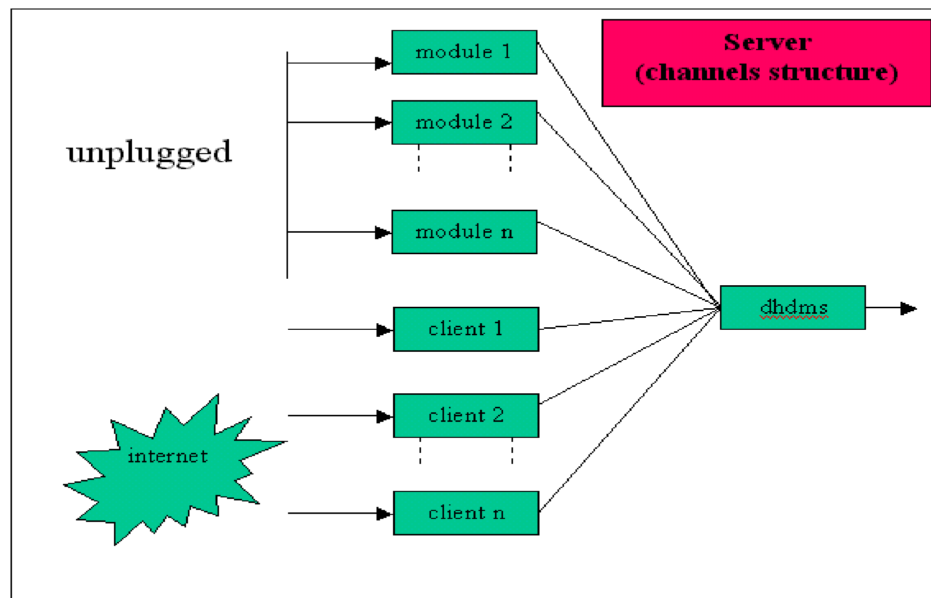
- 0 : indicates that the background bitmap is centered
- `DOCtiled` (4):
  - Indicates that the document background bitmap must be repeated in tile form.
- `DOCstretched` (8):
  - indicates that the document background bitmap needs to be stretched to fit into the document

A main document must be defined for each definition block: this means the '*client*' and '*server*' documents respectively.

To make the download bar appear, a '*download*' zone must be defined in the '*client*' document.

14.6. 6 API

Note that the DMS system creates an unplugged channel for each module, whether this is on the server, the client or the editor. This channel inherits APIs described further on. Consequently, communication between modules always takes place through the API.



14.6.1. Server API

14.6.1.1. Structures

- `DMI` : instance of a module
- `CLIENT` : client
- `User` : user

- `UserI` : user instance
- `Item` : object from the inventory

14.6.1.2. API variables

- `this DMI` : to be used at the start of several of the API's functions
- `DMSsender CLIENT` : indicates the sender of an intra-module message
- `DMSname S` : name of the site
- `DMSwin ObjWin` : server's basic window

14.6.1.3. API functions

➢ **Modules :**

`S _DMSgetName module DMI`
Returns the short name of a module

`S _DMSgetClass module DMI`
Returns the name of a module's Dmc file

`[[S r1] r1] _DMSgetDef module DMI nom_du_bloc S`
Returns a definition block associated with a module

`I _DMSupdateDef module DMI nom_du_bloc S données [[S r1] r1]`
Redefines a definition block (does not save to disk)

`I _DEFsave`
Saves the site

`I _DMScreateClientDMI module DMI client CLIENT param S`
Creates the client instance

`I _DMSdelClientDMI module DMI client CLIENT`
Destroys the client instance

`I _DMSsend module DMI client CLIENT message Comm`
Sends a message to the instance of the same module on a particular client. For this message to be interpreted, there needs to be a function with the same name in the client module, preceded by two underscores.

➢ **Inter-module messages: triggering of events, direct sending of messages**

`I _DMSevent this DMI concerning CLIENT event S param S reply S`
Activates an event by specifying a parameter and a reply. If the parameter is `nil`, the default parameter is used. The reply field should be left at `nil`.

`I _DMSeventTag module user User event S param S others [User r1] [callback fun[param S others [User r1]] I flag timeout]`
Causes an event possibly accompanied by a tag. The flag is unused.

`I _DMStagKeepAlive tag Tag`
Indicates that the tag must be kept even if it is not immediately given as a reply.

I _DMStagForget tag Tag
Indicates that the tag can be forgotten.

I _DMSreplyTag tag Tag param S others [User r1] holdon I
Replies to the tag by passing a parameter to it, and by indicating whether the tag should be destroyed (holdon=0) or if it is going to be used again (holdon=1).

I _DMSdefineActions module DMI liste_actions [[S fun [module_émetteur DMI user User action S param S others [User r1] tag Tag] I] r1]
Defines a list of actions associated with callbacks. This definition is incremental: it can be called more than once, and at any time.

I _DMSremoveActions module DMI actions [S r1]
Removes actions from the list.

I _DMSregister (module DMI) (cb_logout fun [CLIENT] I) (cb_deleteclient fun [CLIENT] I) (cb_beforeclose fun [] I)
I _DMSregisterDMI (module DMI) (cb_action fun [from DMI concerning CLIENT action S param S reply S] I) (cb_deleteclient fun [CLIENT] I) (cb_beforeclose fun [] I)
Records the module's callbacks.

- The `logout` function is called when a client disconnects. The module must then remove any reference to this client.
- The `delete` function is called when a client module is destroyed: either the module has destroyed itself, or the client has disconnected.
- The `beforeclose` function is called before the server module is closed, in other words before shutting down the server.
- The action callback will receive all the action messages received by the module. The new `_DMSdefineActions` function makes this callback obsolete.

➢ **Clients**:

The clients are defined by a CLIENT structure containing mainly:

- A login (which must be unique).
- The reference to the User that has been created to represent the client.
- This User is defined by an ID number that will remain unique and constant throughout the connection.

User CtoU client CLIENT
Returns the user associated with the client.

I _DMSdelClient client CLIENT
Causes the disconnection of a client.

S _DMSgetLogin client CLIENT
Returns the login of a client.

S _DMSsetLogin client CLIENT login S
Modifies the login of a client.

CLIENT _DMSbyLogin login S

Finds a client through its login.

CLIENT _DMSbyLoginI login S
Finds a client through its login, without taking the letter case into account.

I _DMSclientAlive client CLIENT
Returns 1 if the client is still active, 0 otherwise: a module does not keep a reference to a client that no longer exists.

S _DMSgetIP client CLIENT
Returns the IP address of a client.
Each client has a list of variables, called resources variables. The API can be used to define the value of any variable.

S _DMSgetRess client CLIENT ressource S
Returns a resource variable associated with a client.

S _DMSsetRess client CLIENT ressource S val S
Defines a resource variable associated with a client.

➢ Users:

The Users are the mobile entities in the graph. They are of several types:

- client/virtual: corresponds either to a client, or a resident
- global/local: defined for everyone/only on a given client

The Users move in the graph. The services are simple: create/destroy/read data

Each User has a list of items (this was previously managed on the CLIENT structure, the former functions remain valid). This list is an "inventory" containing objects defined by a reference, a name in clear text, a quantity and a date.

User UcreateUser client CLIENT
Creates a global virtual user

I UgetId user User
Returns the ID

CLIENT UtoC user User
Returns the client associated with the user (`nil` if the user is virtual)

I UgetFlag user User
Returns the flag associated with the user

Item _ITEMcreate référence S nom_en_clair S quantité I date I
Creates an item

S _ITEMref item Item
Returns the reference

S _ITEMname Item
Returns the name in clear text

`I _ITEMquantity Item`
Returns the quantity

`I UaddItem user item`
Adds an item to a user

`I UsubItem user référence quantité`
Deletes a particular quantity of an item from the inventory of a user.
Returns the quantity of the item that existed before this deletion (0 if the item was not in the inventory).

`Item UfindItem user chaîne`
Searches for an item in a user's inventory

`I UclearItem user`
Clears all a user's items

➢ **UserInstances**:

UserInstances are a service offered to each module for defining an object linked to a User and providing concepts of distribution, communication and security.

UserInstances are created at server or client level, but a client can only create local instances. Instances are created by specifying:

- a module
- a user
- a class
- parameters
- visibility

The concept of visibility is important: a global instance is only broadcast on clients whose instance associated with this client (i.e., the instance associated with the User corresponding to the client) can 'see' it.

At the moment there is only one type of visibility: by tree. Each instance is placed in a tree. Visibility is then defined by a path in a tree (the list of node names starting from the apex), and a commutativity flag. The rules for visibility are as follows.

- an instance "sees" all the instances that are in its sub-tree (same node or descent)
- if an instance has its commutativity flag set to 1, all instances that it can see can also see this instance.

Similarly, it is possible to isolate an instance.

The services offered for global instances are:

- automatic creation/destruction of a client instance according to the rules of visibility
- transmission of messages between these client instances and the server
- synchronization of the changing of class and parameters

Visibility is defined by its type: **Typedef Visibility :**

- tree [[S r1] flag]

- isolated (nil)

Visibility treeNew rights [S r1] commut I
Returns one Visibility object per tree, by specifying its position in the tree (list of node names starting with the apex) and its commutativity flag.

[UserI r1] Ulist module DMI
Returns the list of the userIs associated with a module

UserI UcreateUI module DMI user User class S parameters [[S r1] r1] visibility Visibility
Creates a user instance. `Nil` as a visibility parameter indicates zero visibility. You will usually use 'treeNew `nil nil'` to define maximum visibility (instance at the apex of the tree)

User UgetUser userI UserI
Returns the associated user

DMI UgetLocation userI UserI
Returns the associated location

[[S r1]r1] UgetParams userI UserI
Returns the associated parameters

[S r1] UgetParam userI UserI champ S
Returns the field's value

UserI UgetUserI module DMI user User
Returns the userInstance associated with a module and a user.

I Udelete userI UserI
Destroys an instance

I UchgClass userI UserI classe S params [[S r1] r1]
Changes the class of a UserI. This will be transmitted to clients that can see the instance and will cause the calling of the callback defined by UcbChanged.

I UsetVisibility userI UserI visibilité Visibility
Changes the rights of a UserI. In general this implies a certain number of UserI creation/destruction commands on the client modules.

I UsetParams userI UserI params [[S r1] r1]
Changes all the parameters. This will be transmitted to clients that can see the instance and will cause the calling of the callback defined by `UcbChanged`.

I UsetParam userI UserI champ S param [S r1]
Changes a field. This will be transmitted to clients that can see the instance and will cause the calling of the callback defined by `UcbChanged`.

UserI UcbClientDestroyed ui UserI callback fun [UserI CLIENT] I
Defines a callback to be informed of the destruction of an instance on a specific client.

serI UcbDelete ui UserI callback fun [UserI] I

Defines a callback to be informed of the destruction of an instance. On the server, this generally occurs when a client disconnects: its userIs are destroyed.

UserI UcbMessage ui UserI liste_de_messages [[S callback fun [UserI CLIENT S S] I] r1]
Defines callbacks on the receipt of messages. These callbacks are simply concatenated with the list present and may mask previous definitions

UserI UremoveMessage ui UserI message S
Removes a callback on a message.

I UsendMessage ui UserI client CLIENT action S param S
Sends a message to a client. If the client is `nil`, the message is sent to all the client instances.

➢ **Zone management**:

[ObjWin I I I I] _DMSgetZone (this DMI, zone S, conflict fun [zone S] I, resize fun [coord [win ObjWin x I y I w I h I] zone S] I, destroy fun [zone S] I)
Requests a zone (the name to be passed is that of the zone defined in the dmi definition blocks).
The `conflict` callback is activated when the zone is requested by another module. The `resize` callback is activated when the zone has changed size. The function returns `nil` if the zone is not associated, and a tuple (main window, x, y, length, height) in the opposite case.

I _DMSreleaseZone (this DMI, zone S)
Releases a zone (the name to be passed is that of the zone defined in the dmi definition blocks).

➢ **Management of documents downloadable by the clients:**

I _RSregister (this DMI, name S, type I, document S)
Saves the document under a particular name, with a specific `'this'` owner. If the type equals 0, the document parameter is the document itself. If the type equals 1, the document parameter is the name of the document file. Transfers will be made from file to memory, without compression.
Returns 0 if OK, -1 if there is an error (document empty or already saved).

I _RSregistersafe (this DMI, name S, type I, document S)
The same as above, but the document will only be accessible to authorized clients.

I _RSregisterfiles(this DMI, files [S r1], type I)
Records a list of files: the name of each document is the name of the file.

I _RSunregister (this DMI, name S)
Withdraws a document

I _RSallowClient (this DMI, client CLIENT, name S)
Gives a client authorization to download a document saved using the function `_RSregistersafe`

I _DMScbUpload module DMI callback fun [CLIENT S S] I
Defines a document receipt callback (_DMSupload function of the client Api). The callback is called when receipt is complete with the following as arguments: client who sent document, name of the document, content.

➢ **Localization functions:**

The Dms architecture integrates a localization kit used to display text messages in different languages. The messages are stored in resource files.

For each module, one resource file per language must be created.

The resource files thus created must be put in a /module lang subdirectory.

The files will respect the following syntax:

- the name of the file takes the name of the module (name of the .dmc file without its extension)
- a first extension indicates the name of the language of the resource file
- (language in English, example: .english, .french,...)
- a second extension, .lang, identifies the resource files.

Example of resource file name: `test.english.lang`

Each line of the resource file corresponds to a reference and to the translation of the corresponding message in the language specified in the file's name. The localization kit is also used to manage messages with parameters. The downloading of files from the server to the clients is automatic. The client retrieves the files in the language of its SCOL Voy@ger. When it changes language, the files corresponding to the new language are automatically downloaded.

On the client, the resource files are stored in the form `NameOfModule.lang` directly in the module's directory.

➢ **Resource File Syntax:**

If you wish to insert comments begin the line with #.

The references are accessible by default both on the client and the server side. To make the references accessible only on the client side, start the name of the reference with an asterisk (do not use the asterisk when the localization function is called, this is automatically removed when the files are loaded into the memory).

Within a message, `\n` is used for the carriage return.

Spaces at the beginning of a message, at the end of a message or two or more spaces in succession will not be taken into account. You can use `\[space]` as a solution to this but it is better to add spaces directly into the program code, as this stops them from being overlooked when the resource files are translated.

Parameters are defined using the syntax `<#no:text>`

- `no` is an integer representing the number of the parameter (which starts at 0)
- `text` is a character string used to specify the nature of the parameter (this string should not contain any spaces: use _ for example as a separator).

This makes it possible to insert one parameter several times. The parameters can be inserted in any order.

➢ **Server Api:**

```
S _loc module DMI reference S parameters [S r1]]
```

Localizes the server in the language of the server SCOL Voy@ger.
Returns the reference's localization (function's 2nd parameter) with parameters inserted (function's 3rd parameter).

`S _locCli module DMI client CLIENT reference S parameters [S r1]`
Used to send the client of a reference localized in its language (the language of the client's SCOL Voy@ger if available, if not the language of the server's SCOL Voy@ger)
Returns the reference's localization (function's 3rd parameter) with parameters inserted (function's 4th parameter). If the reference does not exist in the language of the client, it returns the reference translated in the language of the server.

`S _locCliEx module DMI language S reference S parameters [S r1]`
The same as _locCli, but the language of the client is specified. For example, to send e-mails to a client in its language while it is not connected, the language of the client is specified (function's 2nd parameter) by retrieving it from a server database. Returns the reference's localization (function's 3rd parameter) with parameters inserted (function's 4th parameter). If the reference does not exist in the language of the client, it returns the reference translated in the language of the server.

`I _locAddRef module DMI language S référence S content S`
Used to dynamically add a reference to be localized (function's 3rd parameter) with its content (function's 4th parameter) for a given language (2nd parameter). Returns 1 if added, 0 if not.

`I _locDelRef module DMI language S reference S`
Deletes a reference (function's 3rd parameter) dynamically for a given language (function's 2nd parameter). Returns 1 if deleted, 0 if not.

`I _DMSreinitLoc module DMI`
Dynamically reloads the localization files on the server and on clients that are connected.

In the event of an error, the functions return:

- `"!!ERR_REF!!"` when the reference does not exist
- `"!!ERR_PARAM!!"` when a parameter is missing

➢ **Other services:**

`I _DMStime`
Returns the time of the server

`I _DMStickcount`
Returns the `tickcount` value of the server

`I _DMSservice client CLIENT message S`
Sends a service message to a client. This message appears in a dialog box.

`ObjFont Font`
Site's main font.

`S DMSpath`
Dms file path

S _DMSgetpath nom_fichier S
Returns the path of a file name

[S r1] _DMSrelativpath path S liste_fichiers [S r1]
From a default path and a list of possibly related files (the names of which start with `./`), the function returns a list of absolute file names.

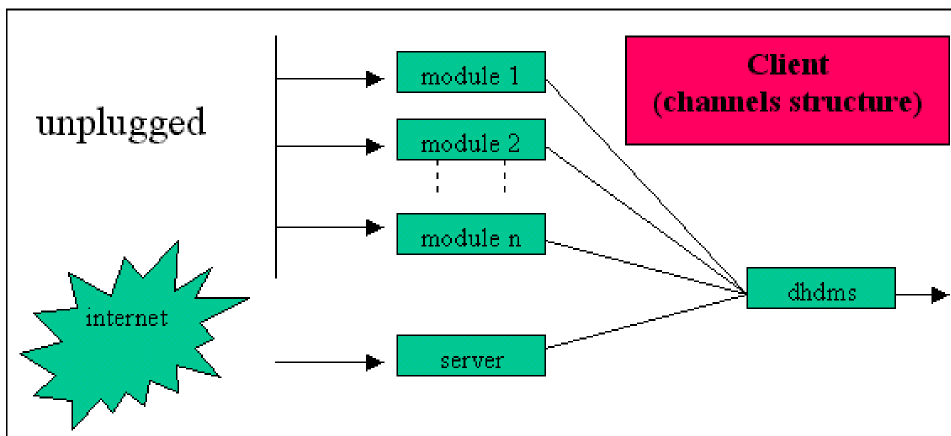
S _adderror message S
Adds an error message to the logbook. If this occurs when the server is started, the server will stop as soon as the initialization of each module is complete.

S _addwarning message S
Adds a warning message to the logbook

S _logBook
Returns the content of the logbook

- **Functions to be defined in the module**

IniDMI (param S)
Function called when the instance is initialized. The parameter that was previously the name of the Dmi file is no longer used.



14.6.2. Client API

14.6.2.1. Structures

- `DMI` : module instance
- `User` : user
- `UserI` : user instance
- `RSC` : download request

14.6.2.2. API variables

- `this DMI` : current instance

- `DMSname S` : site name
- `DMSwin ObjWin` : basic server window
- `DMSlogin S` : client login
- `DMSid I` : client ID number (constant index throughout the connection period)
- `DMSactiveX I` : has the value 1 if the client is an ActiveX (or a Netscape plug-in)

14.6.2.3. 2API constants

- `USER_global`
- `USER_client`
- `USER_changeClass`
- `USER_changeParam`
- `USER_changeAll`

14.6.2.4. API functions

`S _DMSgetName module DMI`
Returns the short name of a module.

`S _DMSgetClass module DMI`
Returns the name of a module's dmc file.

`I _DMSdelete module DMI`
Auto destruction of the client module.

`I _DMSsend this DMI message Com`
Sends a message to the instance of the same module on a specific client.

➢ **Inter-module messages: event triggering, direct sending of messages**

`I _DMSevent this DMI event S param S reply S`
Activates an event by specifying a parameter and a `reply`. If the parameter has the value `nil`, the default parameter is used. The `reply` field should be left at `nil`.

`I _DMSeventTag module event S param S others [User r1] [callback fun[param S others [User r1]] I flag timeout]`
Causes an event, possibly accompanied by a tag. The flag is unused.

`I _DMStagKeepAlive tag Tag`
Indicates that the tag must be kept even if it is not immediately given as a reply.

`I _DMStagForget tag Tag`
Indicates that the tag can be forgotten.

`I _DMSreplyTag tag Tag param S others [User r1] holdon I`
Replies to the tag by passing it a parameter, and indicating if the tag must be destroyed (`holdon=0`) or if it is going to be used again (`holdon=1`).

`I _DMSdefineActions module DMI liste_actions [ [S fun [module_émetteur DMI action S param S others [User r1] tag Tag] I ] r1]`

Defines a list of actions associated with callbacks. This definition is incremental: it can be called more than once and at any time.

`I _DMSremoveActions module DMI actions [S r1]`
Removes actions from the list.

`I _DMSregister (module DMI) (cb_beforeclose fun [] I)`
`I _DMSregisterDMI (module DMI) (cb_action fun [from DMI action S param S reply S] I) (cb_beforeclose fun [] I)`

Saves the module's callbacks.
The `beforeclose` function is called before the server module is closed, in other words before the server is shut down.
The action callback will receive all the action messages received by the module. The new `_DMSdefineActions` function makes this callback obsolete.

- **Users**:

`User UcreateUser`
Creates a local virtual user.

`I UgetId user User`
Returns the ID

`I UgetFlag user User`
Returns the flag associated with the user: this is a combination of the following two constants:

- `USER_global`: global user (as opposed to local)
- `USER_client`: client user (as opposed to virtual)

- **UserInstances**:

`[UserI r1] Ulist module DMI`
Returns the list of the userIs associated with a module.

`UserI UcreateUI module DMI user User class S parameters [[S r1] r1]`
Creates an instance for a local User (this is not widely used).

`User UgetUser userI UserI`
Returns the associated user.

`DMI UgetLocation userI UserI`
Returns the associated location.

`[[S r1]r1] UgetParams userI UserI`
Returns the associated parameters.

`[S r1] UgetParam userI UserI champ S`
Returns the field value.

`UserI UgetUserI module DMI user User`
Returns the userInstance associated with a module and a user.

I Udelete userI UserI
Destroys a local instance.

I UchgClass userI UserI classe S params [[S r1] r1]
Changes a local UserI's class.

I UsetParams userI UserI params [[S r1] r1]
Changes all the parameters of a local instance.

I UsetParam userI UserI champ S param [S r1]
Changes a local instance field.

UserI UcbDelete ui UserI callback fun [UserI] I
Defines a callback to be informed of the destruction of an instance. This generally occurs when the server has called the Udelete function.

UserI UcbMessage ui UserI liste_de_messages [[S callback fun [ui UserI action S param S] I] r1]
Defines callbacks on receipts of messages. These callbacks are simply concatenated with the list present and may mask previous definitions.

UserI UremoveMessage ui UserI message S
Removes a callback on a message.

I UsendMessage ui UserI action S param S
Sends a message to the server instance.

I UcbCreate module callback fun [ui UserI] I
Callback signaling the creation of a global instance on a client.

I UcbChanged ui callback fun [ui UserI flag I valeur S] I
Callback signaling a change of class/parameter for a global instance. The flag can take the following values:

- `USER_changeClass`: the class has changed
- `USER_changeParam`: at least one parameter has changed (if only one parameter, its name is in the value field, otherwise the value field has the value `nil`)
- `USER_changeAll`: equals `USER_changeClass | USER_changeParam`

➢ **Zone management:**

[ObjWin I I I I] _DMSgetZone (this DMI, zone S, conflict fun [zone S] I, resize fun [coord [win x y w h] zone S] I, destroy fun [zone S] I)
Requests a zone (the name to be passed is that of the zone defined in the dmi file).
The conflict callback is activated if the zone is requested by another module. The resize callback is activated if the zone has changed size.

I _DMSreleaseZone (this DMI, zone S)

Releases a zone (the name to be passed is that of the zone defined in the dmi file).

- **Resource download management:**

`RSC _RSCdownload module nom fichier callback fun[S] I salve`
`RSC _RSCdownloadP module nom fichier callback fun[S] I salve priorité I`
Requests the downloading of a resource by giving its name, the file in which the resource must be stored and the callback when the download has finished. The priority parameter indicates a download command: 0 priority requests come first.
The `salve` parameter is obsolete.

If the file is already there, with the right signature, the callback can be called synchronously. The `_RSCdownload` and `_RSCdownloadP` functions return:

- `nil` if the callback has been called,
- otherwise an RSC object: the download is in progress. It is possible to abort it.

If the name's value is `nil`, it is a synchronization download whose callback will be called when all the downloads previously requested (and of lower priority) have been carried out.

If the file's value is `nil`, the resource is still downloaded, is not stored on disk and is passed to the callback, whereas otherwise it is the name of the file that is passed to the callback (`nil` if download is impossible).

`I _RSCabort module ressource RSC`
Aborts the downloading of a resource.

`I _RSCabortDMI module`
Aborts the downloads in progress for a module

`I _DMSupload module DMI nom_document S contenu S callback fun [I] I`
Sends a document to the server by specifying a name and a content. The callback is called once the operation is completed (the argument has a value 1 if successful, 0 if not). This occurs via the HTTP protocol (POST request)

- **Localization functions:**

`S _loc module DMI reference S parameters [S r1]`
Used to localize the client in the language of the client SCOL Voy@ger (the server SCOL Voy@ger language if the client's language is non-existent). Returns the reference's localization (function's 2nd parameter) with inserted parameters (function's 3rd parameter).

In the event of an error, the functions return:

- `"!!ERR_REF!!"` when the reference does not exist
- `"!!ERR_PARAM!!"` when a parameter is missing

- **Other services:**

`I _DMStime`

Returns the server time (approximation)

`I _DMStickcount`
Returns the server's tickcount value (approximation)

`ObjFont Font`
Site's main font

`ObjCursor StdCursor`
Standard cursor

`ObjCursor HandCursor`
Hand cursor

`ObjCursor CrossCursor`
Cross cursor

`S _DMSgetpath nom_fichier S`
Returns a file name path

`[S r1] _DMSrelativpath path S liste_fichiers [S r1]`
From a default path and a list of possibly related files (whose names start with ./), the function returns a list of names of absolute files.

➢ **Functions to be defined in the module**

`IniDMI (param S)`
Function called when the instance is initialized. The parameter is that sent by _DMScreateClientDMI.

14.6.3. Editor API

Writing a module editor is not very complicated: remember that the only purpose of a module editor is to create definition blocks (at least one "dmi" block).

You will use a library called 'templateEdit1' which offers a simple API and a clean and homogeneous graphic interface:

- You must define the `IniEditor` which will be the editor's initialization function
- From the start this function will call the `startEditor` function with the appropriate parameters, and in particular the definition of two callbacks:
  - `load`: callback called when the module is loaded. This function has access to the current values of the module's definition blocks
  - `save`: callback called when saving takes place and which much define the new definition blocks.
- The `IniEditor` function will then initialize the graphic interfaces specific to the module.
- Lastly it will finish by calling the openDMI function which will call the `load` callback.

This API is detailed below:

14.6.3.1. Callable functions

Editor startEditor
  Chn channel
  ObjWin parent
  I *x* /* main window left bound */
  I *y* /* main window upper bound */
  I *w* /* edit window width */
  I *h* /* edit window height */
  I *winflag* /* main window flags */
  I *flag* /* flags for toolbar, statusbar, ... */
  S ___ /* unused */
  S *_class* /* class file */
  S *_help* /* help file */
  S *_icone* /* default icon file */
  fun [[[S r1] r1]] I *load* /* load callback */
  fun [S S] [[S r1] r1] *save* /* save callback */
  [
  [[[I S fun [ObjMenuItem Editor] I] r1] [[I S fun [ObjMenuItem Editor] I] r1]]
  [[[I S fun [ObjMenuItem Editor] I] r1] [[I S fun [ObjMenuItem Editor] I] r1]]
  [[S [ObjMenu r1]] r1]
  ] *additional menus* /* additional menus */

Starts the *"template editor"* with the size of the module's editing zone and the two callbacks, `load` and `save`, as main parameters:

- `load` callback: receives the "Dmi" definition block as a parameter
- `save` callback: must return the "Dmi" definition block

The function returns an **Editor** type object.

`ObjWin getEditWin editor Editor`
Returns the child window of the editor in which the interface for the class can be displayed. From this object, it is easy to obtain the client size and define a callback to resize the editor.

`I setEditorStatus editor Editor message S`
Used to display messages in the editor status bar. This assumes that the corresponding flag was positioned when `startEditor` was called.

`I openDMI  editor Editor`
Loads the module, calling in particular the `load` callback.

`[[S r1] r1] getDef editor Editor name_of_block S`
This function will typically be called in the `load` callback, to read definition blocks other than the "Dmi" block.

`I setDef editor Editor name_of_block S content [[S r1] r1]`
This function will typically be called in the `save` callback to define definition blocks other than the "Dmi"block.

`S _locEditor reference S parameters [S r1]`
Used to localize the editor in the language of the SCOL Voy@ger. Returns the reference's localization (function's 2nd parameter) with inserted parameters (function's 3rd parameter).

14.6.3.2. Functions to be defined

`IniEditor(param S)`

Function called when the editor is started. The parameter is no longer used.

14.7. Example 1 : module running only on the server.

The following example is of a module that manages an automaton able to receive messages, analyze them and reply to them. Usually, this automaton is used by connecting it on a chat cell: the automaton "hears" and can reply either to a single user or to all of them.

File: ***Dms/Bots/Bot0/bot0.dmc***

```
name Bot
serverNeeded ./bot0.pkg
serverLoad ./bot0.pkg
editorNeeded Dms/L/templateEdit0102.pkg locked/lib/_mlistlib001.pkg
locked/lib/enterbox.pkg ./bot0edit.pkg
editorLoad Dms/L/templateEdit0102.pkg locked/lib/_mlistlib001.pkg
locked/lib/enterbox.pkg ./bot0edit.pkg
bitmap Dms/Bots/Bot0/bot0.bmp
tree Dms/Bots/Bot0/bot0.tree.bmp
version 1 2
```

File: ***Dms/Bots/Bot0/bot0.pkg***

```
/* Bot0 - DMS - May 98 - by Sylvain HUET */
/* Mar 00 update */

/* >>>>> PLEASE DO NOT CHANGE THE FOLLOWING FONCTIONS */
fun broad(text)=_DMSeventTag this nil "broad" text nil nil;;
fun private(u,text)=_DMSeventTag this u "private" text nil nil;;
fun user(event,u,text)=_DMSeventTag this u event text nil nil;;

/* >>>>> PLEASE INSERT ONLY HERE YOUR OWN CODE */

/* this function is called when someone says something :
 -u is the visitor who has spoken
 -text is the text the visitor said
*/

fun hear_bot(from,u,action,text,ulist,tag)=
if !strcmp text "foo" then
 (private u "<Bot> bar\n";
  0)
 else if !strcmp text "time" then
 (broad strcat "<Bot> " ctime time;
  0)
 else let hd strextr text -> l in
 if !strcmp hd l "square" then
 (let atoi hd tl l -> x in broad strcat "<Bot> " itoa x*x;
  0)
 else nil;;

/* this function is called when someone is in :
 -u is the visitor who got in
*/
```

```
fun in_bot(from,u,action,param,ulist,tag)=
 broad strcat "<Bot> Hello " _DMSgetLogin UtoC u;
 private u "<Bot> You can ask me for the time or to calculate integer
squares : type 'time' or 'square 4'\n";;

/* this function is called when someone is out :
-u is the visitor who got out
*/
fun out_bot(from,u,action,param,ulist,tag)=
 broad strcat "<Bot> Good bye " _DMSgetLogin UtoC u;;


/* >>>>> PLEASE DO NOT CHANGE THE FOLLOWING CODE */

fun IniDMI(file)=
 _DMSregister this nil nil nil;
 _DMSdefineActions this ["in" @in_bot]::["out" @out_bot]::["hear"
@hear_bot]::nil;;
```

File: ***Dms/Bots/Bot0/bot0edit.pkg***

```
/* Bot Editor - DMS - May 98 - by Sylvain HUET */
/* Rev. Aug. '98 - by Marc BARILLEY */

typeof ed=Editor;;

typeof bannerll=ObjText;;
typeof ll=ObjList;;
typeof addll= ObjButton;;
typeof delll = ObjButton;;
typeof links=[S r1];;

fun updatel2(s,b)=
 _ADDlist ll 1000 s;;

fun updatel()=
 _RSTlist ll;
 apply_on_list links @updatel2 0;;

fun addr(s)=
 if s==nil then nil
 else
 (set links=conc links s::nil;
  updatel);;

fun _add(a,b)=
 iniEnterBox _channel ed.EditorEditWin nil nil "New Event" @addr "Enter a
new event name" ;;

fun _rem(x,b)=
 let _GETlist ll ->[i _] in
 let nth_list links i -> a in
 if a==nil then nil else
 (set links=remove_from_list links a;
  updatel);;
```

```
fun fdlink(a,b)=
 if strcmp hd a "botevent" then 0
 else (set links=conc links (hd tl a)::nil; 0);;

fun load (l) =
  set links=nil;
  apply_on_list l @fdlink 0;
  updatel;
  0;;

fun suppevent(l)=
  if l==nil then nil
  else let l->[a n] in ("botevent"::a::nil)::("event"::a::nil)::suppevent
n;;

fun save (filename, n)=
  ("action"::"in"::nil)::
  ("action"::"out"::nil)::
  ("action"::"hear"::nil)::
  ("event"::"broad"::nil)::
  ("event"::"private"::nil)::
  suppevent links;;

fun rflSizeEditWin (wn, blurp, w, h)=
  _SIZEtext bannerll ed.EditorWEditWin-10 20 5 5;
  _SIZElist ll ed.EditorWEditWin-10 ed.EditorHEditWin-50 5 25;
  _SIZEbutton addll 45 20 5 ed.EditorHEditWin-25;
  _SIZEbutton delll 45 20 70 ed.EditorHEditWin-25;;


fun IniEditor(s)=
  set ed=startEditor
    _channel nil 0 0 315 340 WN_NORMAL-WN_SIZEBOX EDITOR_NORMAL
    s "Dms/Bots/Bot0/bot0.dmc" "Dms/Bots/Bot0/help.txt"
"Dms/Bots/Bot0/bot0.bmp"
    @load @save nil;

  _CBwinSize ed.editWin @rflSizeEditWin 0;

  set bannerll = _CRtext _channel ed.editWin 5 5 ed.wEditWin-10 20
ET_ALIGN_CENTER "New events :";
  set ll = _CRlist _channel ed.editWin 5 25 ed.wEditWin-10 ed.hEditWin-50
LB_DOWN+LB_VSCROLL;
  set addll = _CBbutton _CRbutton _channel ed.editWin 5 ed.hEditWin-25 45
20 0 "Add" @_add 0;
  set delll = _CBbutton _CRbutton _channel ed.editWin 70 ed.hEditWin-25 45
20 0 "Remove" @_rem 0;
  if s==nil then nil else openDMI ed;
0;;
```

This example is deliberately simple.

The Dmc file holds no surprises, it corresponds point for point with what was explained during the description of the Dmc format. You simply need to take note of which files are used by the module and how the scripts are defined.

The bot0.pkg file is at the heart of the program: it works on the server. The `IniDmi` function simply defines the callbacks on the three actions the module recognizes: `in`, `out` and `hear`:

- in: this entry informs the robot that someone has just arrived
- out: this entry informs the robot that someone has just left
- hear: this entry informs the robot that a message has just been "heard"

The module consequently calls three functions, in_bot, out_bot and hear_bot. The idea is this: if the bot module is correctly connected, the in_bot function will be called each time a visitor enters the world; the out_bot function will be called each time a visitor leaves the world; the hear_bot function will be called each time a visitor says something.

These three functions have a User-type u argument. To retrieve a user's name, you will convert the User into CLIENT with the CtoU function and you will use the _DMSgetLogin function, which takes a CLIENT-type as an argument and returns a character string type (see server API)

In the hear_bot function, the text heard is passed in the 'text' argument. To split this argument into words for a syntactic analysis, you simply apply the two 'hd strextr' functions to it. The result is a list of character strings (type [S r1]. Analyze this list, you use the classic functions hd and tl. You can also use the nth_list function (see standard library).

The module defines three functions:

- broad: sends a message to all users
- private: sends a message to a single user
- user: triggers an event concerning a user

You can practice by modifying the code of the in_bot, out_bot and hear_bot functions.

The bot0edit.pkg file illustrates how to define an editor based on functions predefined in the Dms/L/templateEdit0102.pkg file. In this already complex editor example, it is a question of offering to the user the possibility of creating new events. The important function is startEditor. This function creates the editor's standard window (editWin variable), with the File menu, icons, etc. As an argument you give it the name of the Dmc file, the name of the help file and the default bitmap, then two callbacks, load and save.

The load callback is called when the Dmi file is loaded: it takes as an argument the content of the Dmi file to which you have already applied the strextr function. The argument's type is therefore [[S r1] r1].

The save callback is called when the Dmi file is saved. It must return a sequence of lines describing the events, the actions, the zones, and the specific parameters of the module. This result must be the [[S r1] r1] type.

```
startEditor
    _channel nil 0 0 315 340 NORMAL
    s "Dms/Bots/Bot0/bot0.dmc" "Dms/Bots/Bot0/help.txt"
"Dms/Bots/Bot0/bot0.bmp"
    @load @save;
```

The rest of the bot0edit.pkg file manages a list type graphic object, as well as 2 buttons. For this you can use the wEditWin and hEditWin variables, which give the dimensions of editWin.

Exercises:

- **quicksort**: ask the robot to sort a word list
- **calculator**: rather than calculating a square number, take an arithmetical expression (in reverse polish notation to make it simpler)

- **guide**: ask the bot to teleport you somewhere
- **good manners**: automatically teleport a visitor who has said 'shit' to jail
- **successfully take the Turing test.**

14.8. Example 2: distributed module and zone management

The following example is a distributed module. It defines a button on the interface of the client and/or server.

File: ***Dms/Interf/Button/button.dmc***

```
name Button
register ./buttonc.pkg
serverNeeded
serverLoad ./buttons.pkg
clientNeeded
clientLoad ./buttonc.pkg
editorNeeded Dms/L/templateEdit0102.pkg ./buttonedit.pkg
editorLoad Dms/L/templateEdit0102.pkg ./buttonedit.pkg
bitmap ./button.bmp
tree ./button.tree.bmp
version 2 1
```

File: ***Dms/Interf/Button/buttons.dmc***

```
/* Button Server - DMS - march 98 - by Sylvain HUET */
/* Rev. 0101 - Aug. '98 - by Marc BARILLEY */

fun start(from,u,action,param,ulist,tag)=
  if _DMScreateClientDMI this UtoC u nil then
   _DMSeventTag this u "started" nil nil nil
  else nil;;

fun end(from,u,action,param,ulist,tag)=
  if _DMSdelClientDMI this UtoC u then
   _DMSeventTag this u "ended" nil nil nil
  else nil;;

fun IniDMI(file)=
 _DMSregister this nil nil nil;
 _DMSdefineActions this ["start" @start]::["end" @end]::nil;;
```

File: ***Dms/Interf/Button/buttonc.dmc***

```
/* Button Client - DMS - March 98 - by Sylvain HUET */
/* Rev. 0101 - Aug. '98 - by Marc BARILLEY */

typeof button=ObjButton;;

fun pressbut(a,b)= _DMSeventTag this "click" nil nil nil;;

fun _end(s)=
  _DMSdelete this;;
```

```
fun _resizeI(x,s)=
 let x->[win x y w h] in _SIZEbutton button w h x y;
 0;;

fun IniDMI(param)=
 let _DMSgetZone this "Button" @_end @_resizeI @_end ->[win x y w h] in
 if win==nil then nil else
  set button=_CBbutton _CRbutton _channel win x y w h 0 _DMSgetName this
@pressbut 0
 ;;
```

File: ***Dms/Interf/Button/buttonedit.dmc***

```
/* Button Editor - DMS - Mar 98 - by Sylvain HUET */
/* Rev. Aug. '98 - by Marc BARILLEY */

typeof ed=Editor;;
fun save (filename, n)=
  ("action"::"start"::nil)::
  ("action"::"end"::nil)::
  ("eventC"::"click"::nil)::
  ("event"::"started"::nil)::
  ("event"::"ended"::nil)::
  ("zoneC"::"Button"::nil)::
  nil;;

fun IniEditor(s)=
  set ed=startEditor
    _channel nil 0 0 315 125 WN_NORMAL-WN_SIZEBOX EDITOR_NORMAL    s
"Dms/Interf/Button/button.dmc" "Dms/Interf/Button/help.txt"
"Dms/Interf/Button/button.bmp"
    nil @save nil;
  if s==nil then nil else openDMI ed;
0;;
```

Note two important elements in the buttons.pkg file:

- the use of zones in the IniDMI function: `_DMSgetZone this "Button" @_end @_resizeI @_end`
- the creation and destruction of the client module in the server's `start` and `end` functions.

Also note the editor's minimal form: the editor does not actually need a specific user interface.

14.9. Example 3: distributed module and intra-module message

The following example is a module capable of asking the user a question in the form of a message box. We are particularly interested in the exchange of messages between the client module and the server module.

File: ***Dms/Tools/Quizz/quizz.dmc***

```
name Quizz
register ./quizzc.pkg
serverNeeded
serverLoad ./quizzs.pkg
clientNeeded
```

```
clientLoad ./quizzc.pkg
editorNeeded Dms/L/templateEdit0102.pkg ./quizzedit.pkg
editorLoad Dms/L/templateEdit0102.pkg ./quizzedit.pkg
bitmap ./quizz.bmp
tree ./quizz.tree.bmp
version 2 1
```

File: ***Dms/Tools/Quizz/quizzs.pkg***

```
/* Quizz Server - DMS - march 98 - by Sylvain HUET */

defcom Cquizz=quizz S I;;
struct Qz=[cliQz:CLIENT,txtQz:S,numQz:I]mkQz;;

typeof quizz=[S r1];;
typeof qu=[Qz r1];;

fun byboth(a,z)=let z->[c i] in c==a.cliQz && i==a.numQz;;
fun __answer(i,yes)=
 let search_in_list qu @byboth [DMSsender i] -> x in
 if x==nil then nil
 else
 (set qu=remove_from_list qu x;
  _DMSevent this DMSsender strcat if yes then "yes" else "no" itoa i nil
nil);;

fun removecli(l,c)=
 if l==nil then nil else let l->[a n] in
 if a.cliQz==c then removecli n c else a::removecli n c;;

fun logout(cli)=
 set qu=removecli qu cli;
 0;;

fun in(from,u,action,param,ulist,tag,i)=
 let UtoC u-> cli in
 let if i==nil then param else nth_list quizz i -> txt in
 (_DMScreateClientDMI this cli nil;
  _DMSsend this cli Cquizz [txt i];
  set qu=(mkQz[cli txt i])::qu;
  0);;

fun getQuizz(l,i)=
 if l==nil then nil
 else let l->[q n] in
 if !strcmp hd q "quizz" then
 (_DMSdefineActions this [strcat "in" itoa i mkfun7 @in i]::nil;
 (hd tl q)::getQuizz n i+1)
 else getQuizz n i;;

fun IniDMI(file)=
 let _DMSgetDef this "dmi" ->l in
  (set quizz=getQuizz l 0);
 _DMSregister this nil @logout nil;
 _DMSdefineActions this ["in" mkfun7 @in nil]::nil;;
```

File: ***Dms/Tools/Quizz/quizzc.pkg***

```
/* Quizz Client - DMS - March 97 - by Sylvain HUET */
```

```
defcom Canswer=answer I I;;

fun IniDMI(param)=0;;

fun res(x,i,r)=
 _DMSsend this Canswer [i r];;

fun __quizz(s,i)=
  _DLGrflmessage _DLGMessageBox _channel DMSwin "Question" s 2 @res i;;
```

File: ***Dms/Tools/Quizz/quizzedit.pkg***

```
/* Quizz Editor - DMS - feb 98 - by Sylvain HUET */

typeof ed=Editor;;
typeof quizz=tab ObjText;;

fun onequizz(i,x)=
 _CRtext _channel editWin 5 25+i*25 10 20 ET_ALIGN_CENTER itoa i;
 _CReditLine _channel editWin 20 25+i*25 290 20 ET_DOWN+ET_AHSCROLL "";;

fun createQuizz()=
 set quizz=create_tab 8 @onequizz 0;;

fun getQuizz(l)=
 if l==nil then nil
 else let l->[q n] in
 if !strcmp hd q "quizz" then (hd tl q)::getQuizz n
 else getQuizz n;;

fun setQuizz(l,i)=
 if l==nil || i>=8 then 0
 else let l->[a n] in
 (_SETtext quizz.i a;
  setQuizz n i+1);;

/* SCS editor */
fun load (l) =
  setQuizz getQuizz l 0;
  0;;

fun getText(i)=
 if i==8 then nil
 else
 ("action"::(strcat "in" itoa i)::nil)::
 ("event"::(strcat "yes" itoa i)::nil)::
 ("event"::(strcat "no" itoa i)::nil)::
 ("quizz"::(_GETtext quizz.i)::nil)::getText i+1;;

fun save (filename, n)=
   ("action"::"in"::nil)::
   ("event"::"yes"::nil)::
   ("event"::"no"::nil)::
  getText 0;;

fun IniEditor(s)=
  set ed=startEditor
     _channel nil 0 0 315 355 NORMAL
```

```
    s "Dms/Tools/Quizz/quizz.dmc" "Dms/Tools/Quizz/help.txt"
"Dms/Tools/Quizz/quizz.bmp"
    @load @save;
  _CRtext _channel editWin 20 5 290 20 ET_ALIGN_CENTER "Prompt";
  createQuizz;
  if s==nil then nil else openDMI ed;
0;;
```

In this example, the server module sends a message to the client module containing the text of the question as well as a question identifier:

```
_DMSsend this cli Cquizz [txt i];
```

The message is created from the Cquizz communication constructor, defined by:

```
defcom Cquizz=quizz S I;;
```

When the client module receives it, it executes the `__quiz` function. This opens a dialog box with the question's text. When the user replies, the res function is called and transmits the following reply to the server:

```
_DMSsend this Canswer [i r];;
```

The server subsequently executes the `__answer` function, in which the `CLIENT`-type variable `DMSsender` contains the client that sent the message.

14.10. C3d3 Module and plug-ins

14.10.1. Concept of the 3D Api

The C3d module is a very important one: it is capable of managing a 3D space containing animated avatars and objects; it is therefore the most visible module.

It goes without saying that this module manages a 3D scene, but it also offers a system of plug-ins which allows the developer to easily interface new functionalities in the 3D space. This mechanism is based on the system of Users and UserInstances described previously:

- Each of the site's functionalities will be considered as a User:
  - for an object that is turning, you need to imagine that there is a virtual User in the scene whose only role is to make the object turn. This User's parameters are defined by:
    - the object to be turned
    - the values of this rotation (axis, speed, etc.)
- There is therefore one client User per avatar, and one virtual User per functionality
- So that it can use each of these Users, the C3d3 module defines one UserInstance per User. A UserInstance is defined by:
  - a User
  - a name
  - a class
  - various parameters
  - visibility

You will use the class to define the functionality: Avatar, rotation, etc.

The C3d3 module therefore manages a list of UserInstances of different classes. For each class there is a different type of process; a developer must be able to add a new class easily, with a new process.

The concept of the C3d3 plug-in is introduced for this purpose: a C3d3 plug-in is a small program (with a server and/or client part) that manages the operation of a class.

Below we shall define the **Ob** structure, which is a superclass of UserInstances, and explain how to develop a plug-in.

14.10.1.1. Ob structure

The basic structure of the C3d3 module is the **Ob** structure. It represents an "object" in the sense of object programming: in other words an instance of a particular class.

The Ob structure can be used to describe:

- avatars
- various functionalities

The Ob structure is actually a superclass of the UserI structure, and therefore benefits from the mechanism of Users. Typically, avatar Obs will be client User UserInstances, while functionality Obs will be virtual User UserInstances.

When a client enters the module, a UserInstance is created with:

- for the class, its standby value (which will have been previously defined by the `ObSetClass` function)
- for the name, the client's login

When the server is started, the instances that are already present (defined in the editor) are created with:

- the class indicated in the editor
- the name indicated in the editor
- the parameters defined in the editor with two additional lines: `name` and `anchor` (such as they are described in the editor)

The C3d3 integrates the management of the User system's visibility.

Instances other than avatars are always created at the tree's root with the commutativity flag at 1.

The C3d3 module determines the avatar's rights in the following way:

- In the C3d3 editor (advanced menu) you define the resource variable managing rights.
- When the avatar is created, you read this resource variable, which must have the following format:

```
Format strbuild : ((itoa commut) : :rights : :nil) : :nil
```

Where the `rights` parameter is an a.b.c (the point is the separator) type path. The empty string corresponds to the apex of the tree. (a.b.c is the child of a.b).

Tests represent a particular case: if you define the resource variable that manages rights as 'altern' in the C3d3 editor, the avatars will be successively placed in the " 0 " path and the " 1 "path.

Several parameters define the object:

- on the server

  - the UserInstance (and therefore the class, the parameters, and callbacks for communication with the UserClass module and with the client UserInstances)
  - the name

- a destruction callback
- the object's current position (x,y,z)(a,b,c), where this is appropriate

`UserI ObUi(Ob)`
Returns the UserInstance associated with an object.

`I ObMobile(ob Ob)`
Returns the object's mobility flag.

`[I I I] ObPos(ob Ob)`
Returns the object's known position.

`[I I I] ObAng(ob Ob)`
Returns the object's known orientation.

`S ObName(ob Ob)`
Returns the name of the object.

`fun [Ob] I ObCbDestroy(Ob,cb fun [Ob] I)`
Defines a callback to be called before destroying the object.

`[Ob r1] ObList`
Returns the list of objects.

`fun [Ob S] I ObCbSpeak callback fun [Ob S] I`
Redefines the function called on the server on receipt of a chat message.

`[S r1] Obgetglobalress (ress S)`
Returns the value of a C3d resource variable.

`I fun ObSetClass user User class S`
If the user is not already in the 3D cell, indicates the class of the UserInstance that will need to be created when the user appears.
If the user is already in the 3D cell, changes the class of the associated UserInstance.

`I ObUpdateClass user User class S`
If the user is already in the 3D cell, changes the class of the associated UserInstance.

`Ob ObAddInstance class S param [[S r1] r1]`
Creates a new virtual User and a new instance. Among the various parameters, you will use 'name' to determine the name of the instance and 'anchor' for the anchor.

`I ObRemoveInstance id I`
Destroys an instance according to the User's ID number.

It can be useful if a plug-in (server side), having received a given action, decides to " enter " a User in the 3D space in a given position (defined by a name in the C3d3 module editor). Two scenarios are possible:

- the User is already in the 3D cell: you just need to move the user to his new position
- the User is not already in the 3d cell: you need to bring him there

A new function is defined for this:
`I ObPlaceAvatar user User position S`
You pass the User as a parameter as well as the name of the position.
The return value is not important.

- **On the client**

  - the UserInstance (and therefore the class, the parameters, callbacks for communication with the module UserClass and the server UserInstance)
  - the name
  - a mobility flag: does the object's position need to be refreshed and synchronized?
  - an avatar flag: does the object appear in the list of avatars present in the scene?
  - an anchor
  - various callbacks
  - a main 3D object, optional

`UserI ObUi(ob Ob)`
Returns the UserInstance associated with an object.

`S ObName(ob Ob)`
Returns the name of the object.

`I ObAvatar(ob Ob)`
Returns the object's avatar flag.

`I ObMobile(ob Ob)`
Returns the object's mobility flag.

`[Anchor r1] ObAnchor(ob Ob)`
Returns the anchor associated with an object.

`H3d ObSetMain ob Ob objet3d H3d`
Defines the main 3D object.

`H3d ObGetMain(ob Ob)`
Returns the main 3D object.

```
I ObSelect0(id I)
I ObSelect1(id I)
I ObSelect2(id)
I ObSelect3(id I)
```
These functions trigger a 'selectn' client event, with the ID number as a parameter (itoa format)

`I ObSendLocal from Ob to Ob action S param S rep S`
Sends a message to a local instance.

`I ObHear string S`
Outputs a message through the "hear" event.

`I ObSetCam ob Ob`
Defines the object to which the camera is linked: this function links the camera to the main 3D object.

`Surface ObBuffer`
Returns the rendering buffer.

`I ObSetBackground col I`
Defines the rendering's 24-bit background color (`nil: none`).

`[Ob r1] fun ObList`
Returns the list of objects.

The following functions are used to define callbacks: you must not modify the Ob structure yourself.

`fun ObCbGetName(o,f) : fun[Ob] S`
function returning the name of the object

`fun ObCbGetVal(o,f) : fun[Ob S] S`
function returning a particular value

`fun ObCbSetpos(o,f) : fun[Ob [I I I][I I I]] I`
function positioning the object in a particular position

`fun ObCbAnim(o,f) : fun[Ob] I`
function called before each rendering

`fun ObCbSend(o,f) : fun[Ob S S S]`
function called when a message is received

`fun ObCbClick(o,f) : fun[Ob H3d HMat3d I] I`
function called when the user clicks on the main object or one of its descendants (handler, material and button)

`fun ObCbDclick(o,f) : fun[Ob H3d HMat3d I] I`
function called when the user double clicks on the main object or one of its descendants (handler, material and button)

`fun ObCbMove(o,f) : fun[Ob H3d HMat3d] I`
function called when the user passes the mouse on the main object or one of its descendants (handler, material)

`fun ObCbDraw(o,f) : fun[Ob ObjSurface I I I] I`
function tracing the object in 2D on a bitmap, on a given position and size

`fun ObCbControl(o,f) : fun[Ob [[I I I] [I I I]]] I`
function requesting the moving of the object by passing the 2 speed vectors

`fun ObCbControlClick(o,f) : fun[Ob [H3d HMat3d I]] I`
function called each time the user clicks in the 3D

`fun ObCbControlMove(o,f) : fun[Ob [Ob H3d HMat3d]] I`
function called each time the user passes the mouse on a main 3D object. The callback returns two Ob objects: the object that defined the callback, then the object indicated by the mouse.

`fun ObCbControlKeyDown(o,f) : fun[Ob [I I]] I`

function called each time the user presses a key in the 3D

```
fun ObCbControlKeyUp(o,f) : fun[Ob I] I
```
function called each time the user releases a key in the 3D

```
fun ObCbSpeak(o,f) : fun[Ob S] I
```
function called when the user says something

```
fun ObCbPostRender(o,f) : fun[Ob [ObjBitmap [I I]]] I
```
function called after each rendering. The bitmap's value is `ObBuffer()`

```
fun ObCbReceiveLocal(o,f) : fun[Ob Ob S S S] I
```
function called when there is local communication between instances (`ObSendLocal` function)
the callback's parameters are: `from, to, action, param, rep`

```
fun ObCbDestroy(o,f) : fun[Ob] I
```
function called before the destruction of the object

The following functions return the value of the callbacks.

```
fun ObGetName(o) : fun[Ob] S
fun ObGetVal(o) : fun[Ob S] S
fun ObSetpos(o) : fun[Ob [I I I][I I I]] I
fun ObAnim(o) : fun[Ob] I
fun ObSend(o) : fun[Ob S S S]
fun ObClick(o) : fun[Ob H3d HMat3d I] I
fun ObDclick(o) : fun[Ob H3d HMat3d I] I
fun ObMove(o) : fun[Ob H3d HMat3d] I
fun ObDraw(o) : fun[Ob ObjBitmap I I I] I
fun ObControl(o) : fun[Ob [[I I I] [I I I]]] I
fun ObControlClick(o) : fun[Ob [H3d HMat3d I]] I
fun ObControlMove(o) : fun[Ob [Ob H3d HMat3d]] I
fun ObControlKeyDown(o) : fun[Ob [I I]] I
fun ObControlKeyUp(o) : fun[Ob I] I
fun ObSpeak(o) : fun[Ob S] I
fun ObPostRender(o) : fun[Ob [ObjBitmap [I I]]] I
fun ObReceiveLocal(o) : fun [Ob Ob S S S] I
fun ObDestroy(o) : fun[Ob] I
```

```
fun Obgetglobalress(ress S) [S r1]
```
returns the value of a resource variable of the C3d.

You can define clickable objects (i.e., parts of the 3D scene on which the mouse's cursor will be changed and with which move, click double-click callbacks are associated):

```
I ObSetLinks [ob Ob liste_liens [[H3d Hmat3d S ObjCursor fun [Ob H3d Hmat3d
I] I fun [Ob H3d Hmat3d I] I fun [Ob H3d Hmat3d] I] r1]
```

Each link is a tuple containing:

- the link's 3d handler
- possibly the link's material handler (if `nil`, the whole object is a link, independently of the material)
- the link's apparent name

- the mouse cursor to be used (two constants: `HandCursor` (a hand) and `StdCursor` (the simple arrow) can be used)
- `click` callback (arguments: instance, 3d handler, material handler, buttons status)
- `double-click` callback (arguments: instance, 3d handler, material handler, buttons status)
- `move` callback (arguments: instance, 3d handler, material handler)
-

`fun ObGetLinks ob Ob`
returns the previous list

There are some global variables:

`session : S3d`
3d session

`shell : H3d`
the scene's main node

`cam : H3d`
camera

`name3d : S`
name of the cell

When an object is created on the client, there are two possible scenarios:

- **the object corresponds to an avatar**

If it is an avatar other than that of the machine's user:

- The `clickStd` callback is automatically defined:
- `ObSelect1` called on the left button, `ObSelect2` on the right button.

The `setPosStd` position definition callback is automatically defined. It assumes that the avatar has the following structure:

- a shell node representing the avatar's position (usually located on a level with the camera)
- a child 3D object oscillating around this position

If it is the avatar of the machine's user:

The `controlStd` callback is automatically defined:

management of movement with collisions

The `setPosStd` position definition callback is automatically defined. These two functions assume that the avatar has the following structure:

- a shell node placed at the bottom, which turns along the vertical axis
- a collision sphere of one meter's radius located 1m10 from the bottom
- a shell node to which the camera will be assigned, located 1m60 from the feet, and which turns along the horizontal axis

The `speakStd` callback is automatically defined

In addition, if the avatar's class is not present, the 'default' class is used: the aim is to make the avatar appear as early as possible in the scene, even if it is not in its definitive form.

This default avatar (panel with SCOL logo floating in the air) defines the following callbacks:

- animation callback `CbAnim`
- destruction callback `CbDestroy`

When the class is present, the default avatar will be destroyed and replaced by the normal avatar.

As a consequence, you can overload the default avatar by adding a plug-in.

- **the object corresponds to a functionality**

there are no predefined callbacks

14.10.2. Anchors

When developing 3D functionalities, you will quickly notice that there are two types of function:

- Those that are linked specifically to an object, and that will use the 'main object ' field, such as avatars for example.
- Those that require more elements: several 3D objects/materials/positions, such as, for example, a module making several 3D objects follow several trajectories synchronously.

The second type does away with the concept of a main object, which is replaced by that of an anchor. An anchor is an (ordered) list of objects, materials and positions. It is basically a list of Anchor-type elements.

```
typedef Anchor=
            objAnchor [H3d HMat3d S I]
           | posAnchor [S [I I I] [I I I]];;
```

This list is made up of two types of element:

- `objAnchor: a tuple` (3D object, material, name, visibility flag)
- `posAnchor: a tuple` (name of the position, vector, angles)

Thus the anchor is literally the point at which functionality is attached to the 3D scene. For example, a rotation module needs a set of objects to rotate, a movement module needs objects and trajectories, a blinking module needs a list of materials, etc. Some functionalities do not need an anchor: for example, a module displaying a superimposed logo in a corner of the 3D image.

The "`anchor`" parameter of an instance contains the name of the anchor associated with the instance.

This can be the name of an anchor defined in the module's editor, or a direct definition such as:
`strbuild ("#"::nom_H3d::nom_HMat3d::nil)::nil`

In this latter case, the anchor is a list of a single element.

14.10.3. Plug-ins

14.10.3.1. General points

The Ob structure represents an object, in other words the instance of a particular class. The role of plug-ins is to describe classes. There will be precisely one class defined for each plug-in. A plug-in that does not define a class is of no interest.

The plug-in is described by a *.plug file, very similar to the *.dmc format. The same fields can be found:

- `name`: name of the plug-in
- `help`: help text file
- `serverNeeded`: files required by the server plug-in
- `serverLoad`: list of files to be successively complied to start the server
- `clientNeeded`: files required by the client plug-in
- `clientLoad`: list of files to be successively complied to start the client
- `editorNeeded`: files required by the editor plug-in
- `editorLoad`: list of files to be successively complied to start the editor
- `version` version_number subversion_number

Unlike dmc files, here the `server*` lines are optional, as are the `client*` lines. But a plug-in with neither a `server*` line, nor a `client*` line would be of no interest.

14.10.3.2. Internal plug-in

The C3d editor determines the plug-ins required by the instances defined in the editor. When the C3d module is started, the plug-ins required are loaded. N.B.: in order to be recognized by the C3d editor, the directory containing the **.plug* file must be in the *Dms/3d/Plugins* directory.

Once the plug-in has been loaded, the `IniPlug` function is started (the equivalent of the `IniDmi` function for dmi modules), with the name of the **.plug* file as an argument: this file is immediately usable, since its downloading is a prerequisite to the starting of the plug-in.

Typically the plug-in calls the following function:

```
I PlugRegister(class S new fun [Ob] I close fun[] I
```

This function saves the class, with the function called when an object is created, and the function called before the C3d module is destroyed.

This function is the same on the server and on the client.

N.B.: if instances of the class already exist in the module, the '`new`' function will be called before the `PlugRegister` function finishes.

The plug-in has access to a particular API.

it has access to the global variable `'thisplug'`, which is a Plug-type pointer that points to itself (in the same way that a module has access to the variable `'this'`, which is a DMI-type pointer that points to itself).

`[[S r1] r1] PLUGparam plugin Plug`
Returns the plug-in's parameters (types of class variables).

`[Plug r1] PLUGlist`
Returns the plug-in list.

`S PLUGfile plugin Plug`
Returns the name of the .plug file.

`S PLUGclass plugin Plug`
Returns the class of the plug-in.

Plugin information flags: a plug-in may, on the client side, give some information about itself. This generally occurs, once and for all, with the `IniPlug`. These flags are made up of the following masks:

- `PLUGIN_ONLINE_EDITING` the plug-in is intended to be edited online
- `PLUGIN_WHOLE_OBJECT` the plug-in uses an anchor containing a 3D object, with no specified material

The functions for using the information flag are:

`I PLUGinfo plugin Plug`
Returns the current value.

`I PLUGsetinfo plugin Plug`
Defines a new value.

`I PLUGdefineEditor plugin Plug callback fun [ObjWin H3d HMat3d S`
Defines the editor creation callback.

`fun [] S fun PLUGstartEditor plugin Plug window ObjWin 3dhandler H3d material HMat3d parameters S`
Starts the editor associated with a plug-in with initial parameters.

In the editor, the `IniPlug` file is also called, with the name of the **.plug* file as a parameter. The plug-in then typically calls the following function:

`I PlugRegister class S save fun [ [Inst r1] ] [[S r1] [S r1] [[S r1]r1] [[S r1]r1]] close fun[] I openedit fun[ObjWin S] I closeedit fun[] S`
This save function is called just before the file is saved. On input, it recovers the list of instances defined in the editor, whose class corresponds. This function is not called if this list is empty.

The `Inst` structure is defined in the following way:

```
struct Inst=[nameInst:S,classInst:S,anchorInst:S,paramInst:S]mkInst;;
```

The save function must return a tuple of two lists of words, and two lists of word lists, which makes a tuple of four elements:

- the first element is a list of files to be added to the `'registerF'` line in the dmi block
- the second element is a list of files to be added to the `'register'` line in the dmi block
- the third element is a list of lines to be added to the end of the dat block (`strextr` format)
- the fourth element is a list of lines to be added to the end of the dmi block (`strextr` format)

N.B.: the **.plug* and *clientNeeded* files are automatically saved as downloadable files: there is no point in returning them in the save function.

Generally, you will just need to add the line `'plugin file_*.plug'` to the dat block.

You will be able to add dmi new elements, new actions and new zones to the dmi file.

If the save function is not defined (`nil` in the `PlugRegister` function) a standard `save` function will be called, which returns the following tuple:

```
[
  nil /* registerF */
  nil /* register */
  ("plugin"::plugin_file::nil)::nil /* supplemental Dat */
  nil /* supplemental Dmi */
 ]
```

Most of the time this suffices.

14.10.4. Examples

14.10.4.1. Example 1: rotate module

In this example, the "long" version of the editor is given, a version equivalent to if there was no definition of the 'save' function.

file ***Dms/3d/Plugins/Rot/rot.plug***

```
name Rotate
help Dms/3d/Plugins/Rot/rot.help
clientNeeded ./rotc.pkg
clientLoad ./rotc.pkg
editorNeeded ./rotedit.pkg
editorLoad ./rotedit.pkg
version 2 0
```

File ***Dms/3d/Plugins/Rot/rotc.pkg***

```
/* Rotate Plugin - DMS - March 00 - by Sylvain HUET */

typeof class=S;;

fun rotobj2(x,v)=
 match x with
  (objAnchor [h _ _ _] -> M3rotateObj session h v)
 |(_->nil);;

fun rotobj(o,v)=apply_on_list ObAnchor o @rotobj2 v;;

fun newOb(o)=
 let hd UgetParam ObUi o "angular" -> s in
 let nth_char s 0 -> a in
 let if (a>=48 && a<58)||a=='- then ['y 109*atoi s]
  else [a 109*atoi substr s 1 1000] ->[v i] in
 let if v=='x then [0 i 0]
  else if v=='z then [0 0 i]
  else [i 0 0] -> v in
   ObCbAnim o mkfun2 @rotobj v;
 0;;

fun IniPlug(file)=
 set class=getInfo strextr _getpack _checkpack file "name";
 PlugRegister class @newOb nil;
 0;;
```

File ***Dms/3d/Plugins/Rot/rotedit.pkg***

```
/* rotedit.pkg : editeur du plugin rot */
```

```
typeof plugin=S;;

proto save=fun [ [Inst r1] ] [[S r1] [S r1] [[S r1]r1] [[S r1]r1]];;

fun save(l)=
[
  nil /* registerF */
  nil /* register */
  ("plugin"::plugin::nil)::nil /* supplemental Dat */
  nil /* supplemental Dmi */
 ];;

fun IniPlug(file)=
 set plugin=file;
 PlugRegister (getInfo strextr _getpack _checkpack file "name")
  class nil nil nil nil;;
```

14.10.4.2. Example 2: test module

The test module demonstrates the server plug-in and the possibilities for communication between server object and client object. The principle is the following: by clicking on an object (first element of the instance's anchor), the user randomly changes the object's flat color. This change is global: it applies to everyone. It is also persistent: the server permanently keeps the current color, and transmits it to new clients.

The message system is the following:

- the server transmits the color with the `'setFlat'` message
- the client requests the current color with the `'color?'` message
- the client indicates that it is clicking on the object with the `'click'` message

The communication functions used are those of UserInstances, used as superclasses of the Ob structure.

File ***Dms/3d/Plugins/Test/test.plug***

```
name Test
help Dms/3d/Plugins/Test/test.help
serverNeeded ./tests.pkg
serverLoad ./tests.pkg
clientNeeded ./testc.pkg
clientLoad ./testc.pkg
editorNeeded ./testedit.pkg
editorLoad ./testedit.pkg
version 2 0
```

File ***Dms/3d/Plugins/Test/tests.pkg***

```
/* Rotate Plugin - DMS - March 99 - by Sylvain HUET */

typeof class=S;;
```

```
fun cbcomm(ui,cli,action,param,z)=
 let z->[o col] in
 if !strcmp action "click" then
 (set col=(rand&255)+((rand&255)<<8)+((rand&255)<<16);
  mutate z<-[_ col];
  UsendCli this nil ui "setFlat" itoa col)
 else if !strcmp action "color?" then
  UsendCli this cli ui "setFlat" itoa col
 else nil;;

fun newOb(o)=
 UcbComm this ObUi o mkfun5 @cbcomm [o 1];
 0;;

fun IniPlug(file)=
 set class=getInfo strextr _getpack _checkpack file "name";
 PlugRegister class @newOb nil;
 0;;
```

File ***Dms/3d/Plugins/Test/testc.pkg***

```
/* Rotate Plugin - DMS - March 99 - by Sylvain HUET */

typeof class=S;;

fun appFlat(x,col)=
 match x with
  (objAnchor [_ m _ _] -> M3setMaterialFlat session m col)
 |(_->nil);;

fun applyFlat(o,col)=
 apply_on_list ObAnchor o @appFlat col;
 0;;

fun cbcomm(ui,action,param,o)=
 if !strcmp action "setFlat" then
  applyFlat o atoi param
 else nil;;

fun cbclick(o,h,m,i)= UsendSrv this ObUi o "click" nil;;

fun newOb(o)=
 UcbComm this ObUi o mkfun4 @cbcomm o;
 match hd ObAnchor o with
  (objAnchor [h _ _ _] -> ObSetMain o h)
 |(_->nil);
 ObCbClick o @cbclick;
 UsendSrv this ObUi o "color?" nil;
 0;;

fun IniPlug(file)=
 set class=getInfo strextr _getpack _checkpack file "name";
 PlugRegister class @newOb nil;
 0;;
```

File ***Dms/3d/Plugins/Test/testedit.pkg***

```
/* edit.pkg : editeur du plugin */
```

```
fun IniPlug(file)=
 PlugRegister (getInfo strextr _getpack _checkpack file "name")
  class nil nil nil nil;;
```

_masterchannel..92
3D......60, 61, 62, 63, 66, 68, 73, 75, 102, 104, 106, 107, 108, 109, 138, 144, 145
3D objects 60
ActiveX...112, 124
BigNum...77, 78
C3d...138
callback 13, 14, 57, 116, 120, 125, 126, 133
cameras 60, 63, 65
channel ...16, 18, 40, 42, 43, 44, 45, 46, 47, 48, 49, 55, 56, 58, 59, 89, 90, 92, 93, 100, 109, 114
collision.........................60, 62, 63, 74, 75, 76, 145
communication constructor..........20, 22, 47, 138
condition 23, 29, 78, 104, 105, 112
console......................9, 10, 16, 34, 40, 81, 89, 103
cookies 53, 54
dmc....108, 111, 130, 132, 133, 134, 135, 136, 138
dmi 111, 112, 120, 126
DMS..........102, 103, 108, 114, 134, 135, 136, 137
documents 104, 106, 112, 120
editor8, 63, 103, 106, 107, 108, 109, 111, 114, 128, 129, 133, 135
environment6, 7, 15, 16, 18, 42, 43, 44, 45, 47, 49, 90, 93
events 46, 56, 57, 74, 105, 111, 115, 124, 133
files....6, 7, 9, 10, 32, 41, 42, 45, 46, 51, 52, 53, 54, 56, 59, 60, 62, 63, 65, 66, 67, 77, 89, 91, 93, 94, 107, 108, 109, 111, 120, 128, 133
Hello World 9, 10, 11, 12, 13, 17, 42, 56, 57
inference of type 11, 17
M3D...63, 66, 67, 68, 69
material 61, 62, 63, 65, 66, 68, 71, 72, 73, 145
memory ...6, 36, 48, 62, 68, 69, 89, 90, 91, 92, 107, 108, 109, 111, 120
modules61, 102, 103, 104, 105, 106, 107, 108, 109, 111, 112, 114, 115, 124
multimedia..87, 102
operating rights 91
parentheses...10, 16, 49
partition....................................6, 7, 9, 51, 52, 107
plug-in..80, 124
polymorphic..17, 27, 32
polymorphism 18, 19, 20
proprietary channel 56
quicksort...27
recursion 18, 19, 20, 27, 113
redefinition..32
reflex...57, 58, 59
Scene..60, 62
script .10, 42, 43, 44, 45, 46, 47, 49, 52, 89, 90, 91, 92, 93, 94, 97, 99, 100, 102, 108, 109
scs 112
server 18, 43, 44, 45, 46, 47, 48, 53, 89, 93, 94, 102, 103, 104, 105, 106, 107, 108, 109, 111, 113, 114, 115, 116, 123, 125, 130, 133, 134, 136, 138
session...........61, 66, 67, 68, 69, 70, 71, 72, 73, 75
side effects 16, 20
signature...................................37, 52, 53, 54, 107
signing...34
SQL...80, 81, 82
standard client...53, 93, 94
standard server 93, 94
startup 107
start-up 10, 16, 45, 51, 52, 89, 90, 92, 93, 109
structures...29, 114, 123
syntax 7, 8, 10, 18, 20, 42, 43, 45, 49, 65, 90
tables 19, 28, 34, 38
TCP/IP 42, 43, 44, 48, 49, 89, 113
timers 56, 59
tuples....................................18, 19, 25, 26, 27, 32
TUTORIAL LANGAGE SCOL 1
type constructors 29, 30
types..10, 17, 18, 19, 20, 29, 32, 42, 46, 52, 57, 62, 63, 64, 68, 74, 80, 104, 105, 107, 108, 145
UDP 42, 44, 48, 49
variable6, 13, 15, 18, 20, 22, 23, 25, 29, 30, 32, 63, 68, 94, 104, 117, 133, 138
virtual machine.....................................6, 7, 10, 90