

The Scriptol Programming Language

Complete reference manual for the version 7.0 of Scriptol.

The language is entirely implemented into the Scriptol to C++ compiler.

- Embedding inside Web page is possible only with the Scriptol to PHP compiler.
- The XML data-structure is not available in the scriptol-php compiler for now.
- The interpreter at scriptol.com is limited and recognizes only a subset of the language for now (see at the change page for details).
- XML is handled by the interpreter as static and dynamic data-structure.
- XML is handled by the compiler (C++) as dynamic classe named "dom" described in the "dom.html" file. This is compatible with the format of the interpreter (see examples on scriptol.com). In a near future, the XML format of the interpreter will be extended to the two compilers.

Mail: webmaster@scriptol.com

Home page: <http://www.scriptol.com/>

Node:Top, Next:[Overview](#), Previous:([dir](#)), Up:([dir](#))

Table of Contents

- [Overview](#)
- [About this manual](#)
- [Interpreting a Scriptol program](#)
- [Compiling a Scriptol program to PHP](#)
- [Compiling a Scriptol program to native](#)
- [My first program](#)
- [Scriptol project](#)
- [Scriptol source file](#)
- [Features of the language](#)
- [Scriptol in html page](#)
- [Statement](#)
- [Comment](#)
- [Symbols](#)
- [Identifiers and keywords](#)
- [Variables or primitives](#)
- [Literals](#)
 - [Quotes and escaping](#)
 - [Variable in string](#)
- [Declaration](#)
- [Constant](#)
- [Nil and null](#)
- [Assignment](#)
 - [Simple assignment](#)
 - [Compound assignment](#)
 - [Multiple assignments](#)
 - [Conditional assignment](#)

- [Operators](#)
- [Expression](#)
 - [Precedence](#)
- [Function](#)
 - [Alias: Arguments of a function](#)
 - [Default values](#)
 - [Scope and function](#)
 - [The quiet mode](#)
 - [The "main" function](#)
- [Print and echo](#)
 - [Input](#)
- [Control structures](#)
- [If](#)
 - [Composite if](#)
- [For in](#)
- [Scan by](#)
 - [Scanning a two-dimensional array](#)
- [While](#)
 - [While let](#)
- [Do until](#)
- [Do case](#)
- [Do extended syntax](#)
- [Break and continue](#)
- [Enum](#)
- [Indexing](#)
- [Range](#)
- [Subscripting](#)
- [Text](#)
 - [Methods on text](#)
- [Dynamic variables](#)
- [Sequence and list](#)
- [Array](#)
 - [Making an array](#)
 - [Indexing an array](#)
 - [Iterator](#)
 - [Using array as a stack](#)
 - [Interval in array](#)
 - [Operators on array](#)
 - [Multi-dimensional array](#)
 - [Content of PHP array, step by step](#)
- [Dictionary](#)
 - [Making a dict](#)
 - [Indexing a dict](#)
 - [Interval and dictionary](#)
- [Methods of array and dict](#)
- [Typed arrays](#)
 - [Constructor and literal](#)
 - [Array of int](#)
 - [Array of text](#)
 - [Array of real, natural, number](#)

- [Array of objects](#)
 - [Assignment and conversion](#)
 - [Using typed array with dyn](#)
 - [Random assignment](#)
 - [Limitations and compatibility](#)
 - [File](#)
 - [Read](#)
 - [Write](#)
 - [Dir](#)
 - [The error control structure](#)
 - [Scopes](#)
 - [Extern, external variables, constants and functions](#)
 - [Variables and constants](#)
 - [External functions](#)
 - [External classes](#)
 - [External types](#)
 - [Inserting native code](#)
 - [Class](#)
 - [Defining a class](#)
 - [Constructor and instance](#)
 - [Static methods and attributes](#)
 - [Inheritance](#)
 - [Overloading](#)
 - [Static XML](#)
 - [Light XML](#)
 - [Defining an XML document](#)
 - [Loading and saving](#)
 - [XML assignment](#)
 - [XML iterator](#)
 - [Using data](#)
 - [Using attributes](#)
 - [XML in functions](#)
 - [Dynamic XML](#)
 - [XML methods](#)
 - [Included files](#)
 - [The define statement](#)
 - [Using a function as a variable](#)
 - [Using an external type](#)
 - [Using standard libraries](#)
 - [Using PHP libraries](#)
 - [Using C libraries](#)
 - [Useful functions](#)
 - [Integrated XML \(Scriptol C++ only\)](#)
 - [Appendix I: Using the Java API with Scriptol](#)
 - [Appendix II: Exceptions handling](#)
 - [Appendix III: Foreign programming](#)
 - [Appendix IV: Deprecated syntax](#)
 - [Index](#)
-

Node:Overview, Next:[About this manual](#), Previous:[Top](#), Up:[Top](#)

Overview

The Scriptol programming language may be used with interpreters and compilers.

A Windows and Unix version exist for each compilers.

The goal of the Scriptol language is to be simple, natural and thus to reduce risk of errors. Scriptol means to "Scriptwriter Oriented Language". It is an universal language designed to produce dynamic web pages, scripts and GUI based applications. It may be a good scripting language for tools that produce XML documents.

Scriptol may be embedded inside a html page and is converted to the same page with PHP code, ready for the Net and fully portable.

The language has been defined according to seven rules displayed on the scriptol.com site, and a page explains also why you must use Scriptol.

Node:About this manual, Next:[Interpreting a Scriptol program](#), Previous:[Overview](#), Up:[Top](#)

About this manual

Please note that the [] symbols included into the syntax of statements are usually not part of the syntax itself and denote an optional item, but for indexing and intervals.

Node:Interpreting a Scriptol program, Next:[Compiling a Scriptol program to PHP](#), Previous:[About this manual](#), Up:[Top](#)

Interpreting a Scriptol program

The command is:

si [options] sourcefile

If errors are encountered, they are displayed, and the program is not executed.

Options of the interpreter:

- none: interpret and execute a scriptol source file.
- t: test only, don't execute the code.
- v: verbose, display detailed infos.
- q: quiet, no message at runtime.
- f: force, ignore errors and run the program.

Node:Compiling a Scriptol program to PHP, Next:[Compiling a Scriptol program to native](#), Previous:[Interpreting a Scriptol program](#), Up:[Top](#)

Compiling a Scriptol program to PHP

Under Windows, the command is:

```
solp [options] sourcefile
```

If the file is already compiled, the program is directly launched, else it is compiled first, and if no error is encountered, it is executed.

While the source has errors, the command "solp sourcefile" compiles it again. If the source is a html page, it is not executed after compilation.

Under Linux, the command to compile is:

```
solp [options] filename
```

and the command to run the php code is:

```
php -q sourcefile.php
```

Options of the Scriptol-php compiler:

```
none:  run a scriptol file, compile it if needed.
-w:    compile code embedded inside html page and make a php file.
-r:    forces to run. Compile if needed and invoques the interpreter.
-b:    recompile the file and all included ones. Don't interpret.
-t:    invoques translation of the source before compiling.
-v:    displays more messages when running a program.
-q:    no message after compilation.
-4:    compile to the syntax of PHP 4 (classes of PHP 5 are not available).
```

These letters may be combined in a single option.

Ex: -be is equivalent to -b -e

Node:Compiling a Scriptol program to native, Next:[My first program](#), Previous:[Compiling a Scriptol program to PHP](#), Up:[Top](#)

Compiling a Scriptol program to native

The command is:

```
solc [options] mainfile
```

The main file is the one that holds the main() function. Dependency are calculated by the compiler which knows what to compile or not.

Options of the Scriptol-C++ compiler:

```
none:  compile only that must be compiled.
-b     build all object files.
-c     compile all sources into C++ only.
-e     make executable.
-r     run the executable.
-t     invoques translation before compiling.
-v     more messages.
-q     no messages.
```

Node:My first program, Next:[Scriptol project](#), Previous:[Compiling a Scriptol program to native](#), Up:[Top](#)

My first program

A Scriptol source is just a text file with commands inside, one per line.

Examples:

```
print x * 2
print x + (y / 2)
```

With the text editor included in the archive, or any other text editor, enter this text:

```
print "Hello world"
```

Save it into the hello.sol file.

Click on the "interpreter" or "interpret" command in the "Tools" menu, the "Hello world" sentence will be displayed.

If you are working in a command line windows (called MS-DOS under Windows or console under Unix), just type:

```
si hello          or
solp hello
```

You may also build and run an executable with the commands:

```
solc -bre hello          build hello.exe or hello under Unix
hello                   run hello.exe or hello
```

You can work both with the editor and a command-line windows.

Node:Scriptol project, Next:[Scriptol source file](#), Previous:[My first program](#), Up:[Top](#)

Scriptol project

No project file is required to build a whole Scriptol application.

Just as Java or PHP, you can compile an entire project just by compiling the main source file.

A Scriptol source must have an "include" statement for each file the content of which it uses. The compiler calculates the dependancies among all levels of embedding and solves cross including.

Node:Scriptol source file, Next:[Features of the language](#), Previous:[Scriptol project](#), Up:[Top](#)

Scriptol source file

A Scriptol file must have the ".sol" extension, and may be converted into a newer file with either the ".php" or ".cpp" and ".hpp" extensions or binary file with a ".exe" extension under Windows and no extension under Unix.

Sources files may be either scripts or applications (or dynamic web pages).

A script source is a list of statements along with definitions of functions or classes...

A script, a source with statements at global level, can't hold a "main" function because for the compilers, the whole script is itself a "main" function.

An application source holds only functions and data declarations. The main file is the source argument of the compilation command, a "main" function must be defined into the main file, this function must be called, as following:

```
int main()
    ... statements ...
return 0

main()                ...starting the program
```

Node:Features of the language, Next:[Scriptol in html page](#), Previous:[Scriptol source file](#), Up:[Top](#)

Features of the language

The design of the Scriptol language has been made with a rigorous method and not empirically by adding features from time to time, when needed. Seven rules have been defined to lead the design (look at the "seven" page on the site), and nothing has been included into the language that doesn't satisfy all the rules. I decided as first rule that the language must be simple and natural, and the second one is to suppress all causes of errors for the programmer.

Scriptol may be defined as:

- object-oriented.
- XML oriented (XML document may be a data-structure in source).
- universal: usable for scripting, dynamic web pages, building executables.
- natural: types of variables come from sciences and not from hardware: number, text, real...
- XML-like styled syntax.
- featuring new and very powerful control structures.
- list processing on arrays and dictionaries.
- PHP, C++ and Java compatible.

It is a clear language thanks to:

- a simple syntax.
- statements ended by end of lines.
- a same operator for ranges, slices, splices...
- a similar syntax for all structures.

Case-sensitivity:

- You cannot use any word both in lowercase and uppercase.
- Keyword must be lowercase.
- Identifiers are case-sensitive, but you can't redefine an identifier with different case.

Identifiers:

- size up to 255 chars or less according to the target language.
- lower or upper case.
- start with a letter, continue with letters, underscores or digits.

Numbers:

- int are signed 32 bits. (as "int" in C).
- naturals are 64 bits unsigned.
- reals, numbers are 64 bits floating-point. ("double" in C)

Cast:

- casting by use of methods.

Garbage-collector:

- automatical memory management, no need to allocate and free memory.

Object-oriented:

- primitives are objects and have methods.
- literals are objects and have methods.
- single-inheritance.
- overloading of methods (in Scriptol C++ only for now).
- constructors. No destructors.

XML oriented:

- XML documents may be included into Scriptol sources. XML is a data structure of the language.
- instances of XML documents.

Node:Scriptol in html page, Next:[Statement](#), Previous:[Features of the language](#), Up:[Top](#)

Scriptol in html page

For Scriptol code embedded inside html, it should be inserted inside the following tags:

```
<?sol
    ...code...
?>
```

If you use a html editor that doesn't recognize these markers, use the following ones instead:

```
<script language="scriptol">
    ...code...
</script>
```

Writing 'scriptol' or "scriptol" or scriptol is valid.

Inside theses markers, the last line of a script must be terminated by a semi colon or an end of line (before the ?> symbol).

The simplest way to use a Scriptol script is to call it from a PHP page, that holds a simple include statement:

Example:

```
<?php
    include_once("count.php");
    update("manual.dat");
?>
```

This example calls a PHP counter inside the count.php file, built from the count.sol file with the command:

```
sol -w count.sol
```

Testing html pages

If you have installed a server as Apache or Xitami or Windows Server on your computer, and configured them to recognize the php extension, your code will be processed as on a the web, once compiled to PHP.

Otherwise you have to redirect the output of the script into a html file, "test.html" for example, with two commands that produce mypage.php, and execute the PHP code:

```
solp -w mypage  
php mypage.php > test.html
```

Node:Statement, Next:[Comment](#), Previous:[Scriptol in html page](#), Up:[Top](#)

Statement

A statement is ended by a semi-colon or by the end of the line.

When a statement exceeds the width of a line, the line is concatenated with the following one providing that it is ended by a comma, or by an operator. No symbol is required to continue a statement on the next line.

Multiple statements on a same line are separated by a semi-colon. This is provided mostly to leave freedom to programmers, but may be useful if you insert Scriptol code into an html page and the editor concatenates the lines!

Node:Comment, Next:[Symbols](#), Previous:[Statement](#), Up:[Top](#)

Comment

Comments are destined to the reader only and are skipped by the interpreter.

A one line comment start with the ` symbol, the text is skipped up to the end of the line.

The `` doubled symbol makes a comment persistent into generated code when the Scriptol source is compiled into another language as C++ or PHP .

```
` this is a full line comment  
print x      ` this is a comment at end of line  
`` this is a persistent comment
```

The // of C++ is also recognized a full line comment.

A multi-lines comment starts with /* and ends with */. The enclosed text is skipped.

Example:

```
/*  
    Inside these markers, anything is ignored by the compiler
```

Node:Symbols, Next:[Identifiers and keywords](#), Previous:[Comment](#), Up:[Top](#)

Symbols

The Scriptol language does not use a symbol for things that are conceptually different .

```
"+" "-" "*" "/" are arithmetic operators you know.
"=" operator of assignment.
"<" less than.
">" greater than.
"<=" less or equal.
">=" greater or equal.
":=" conditional assignment.
";" is a end of statement terminator.
"," is a separator. It separates elements of an initializer, or of a tuple.
":" builds an association. It attaches a body to a header, a value to a key, a method to an object, and so one...
"." uses an association. It associates a method call to the object.
".." separates the limits of an interval of two numbers.
"--" is also an interval symbol, upper limit beeing not included.
"()" groups sub-expressions, or arguments of a function.
"[]" denotes an index or an interval.
"?" terminates a condition in one-line control structures.
"<>" not equal
"&" binary and, or intersection of arrays.
"|" binary or, or union of arrays.
"<<" shift left.
">>" shift right.
"and" logical and (on boolean values).
"or" logical or.
"not" logical negation.
"mod" modulo ("% from C++ is not recognized).
"~~" encloses code to be inserted directly into the target language.
```

Node:Identifiers and keywords, Next:[Variables or primitives](#), Previous:[Symbols](#), Up:[Top](#)

Identifiers and keywords

Identifiers are names of variables, functions or objects. They start with a letter, followed by letters, underscores or digits.

They are not case-sensitive, you can't give same name to two different objects, one in uppercase and the other in lowercase.

Keywords are reserved words of the language and are lowercase.

Node:Variables or primitives, Next:[Literals](#), Previous:[Identifiers and keywords](#), Up:[Top](#)

Variables or primitives

Variables are names associated at memory fields holding scalar values. Primitives are basic variable implemented in the language.

Scriptol's primitives are these:

number	any king of number (up to "double" in C++).
int	a number rounded to the integer part. 32 bits. -1 is nil.
natural	64 bits unsigned integer. 0 is nil.
real	a number with decimals. -1 is nil.
boolean	the true or false value.
text	a string of characters. "" is nil.
array	an indexed and dynamic list of objects
dict	an associative list of pairs key:value.
dyn	generic element of array or value of dict.
file	a file.
dir	a directory.

Other types have been added to use external C extensions:

byte *	external C type.
cstring	used in C++ or Java declarations (converted to char *).
*	address of variable, used to declare callbacks from C functions.

Basic objects have constructors as other objects:

Syntax:

```
type (value)
```

Examples:

```
int ("123")
```

represents the 123 value.

```
text (10)
```

represents "10".

Constructors of scalars are mainly used to convert from a type to another.

```
text t = "10"  
int i = int(t)
```

Node: [Literals](#), Next: [Quotes and escaping](#), Previous: [Variables or primitives](#), Up: [Top](#)

Literals

A text is written: "sometext" or 'sometext'.

In the first case, symbols inside the string as \$ or { have special meanings when the target language is PHP.

You can span a text literal in several lines.

Example:

```
print "first line"
      "second line"
```

Literals numbers are written:

```
123          integer (int)
123n         natural
123.0        real
0.123        real
1.2e10       real
0xf8         hexadecimal
true/false   boolean
```

Node:Quotes and escaping, Next:[Variable in string](#), Previous:[Literals](#), Up:[Top](#)

Quotes and escaping

A literal text may be enclosed into simple or double quotes.

```
print "abc$xyz"
print 'abc$xyz'
```

In the first case, this displays a string of these characters: abc followed by the content of the xyz variable.

In the second case, this displays this string: abc\$xyz.

To insert characters that cannot be directly inserted with the keyboard, the special escape code `\` is used, followed by a letter:

```
\ "      inserts a double quote
\'      inserts a single quote
\n      inserts a newline.
\t      inserts a tab.
\\      inserts the \ character itself.
```

Multi-lines

A string may be shared on several line when enclosed inside `"~~"` markers.

Example:

```
x = ~~
   row 1
   row 2
   ~~
```

Line feed may be either direct or escaped:

```
x = "one\ntwo"           ...valid
x = "one
two"                     ... not valid

x = ~~one
two~~                     ... valid
```

Node:Variable in string, Next:[Declaration](#), Previous:[Quotes and escaping](#), Up:[Top](#)

Variable in string

Inside a PHP-like string some symbols have special meaning at runtime, a \$ symbol means for a variable name.

```
text xyz = "def"  
text t = "abc$xyz "
```

The t variable is assigned the characters a,b,c followed by the content of the variable xyz, and thus, t really holds the string: abcdef.

In the same manner, the {} symbols have special meaning for the PHP target, but this is not part of the Scriptol language.

The recommended syntax is rather:

```
text t = "abc" + xyz
```

A string inside simple quotes and prefixed by \$ is not interpreted at all.

```
text t = 'av$er\n'
```

is displayed as: av\$er\n

Node:Declaration, Next:[Constant](#), Previous:[Variable in string](#), Up:[Top](#)

Declaration

The declaration of a variable has the form:

```
type var [ = expression ]  
int x  
text t = "demo"
```

Declaration of several variable at once is allowed in the form:

```
type name [= value], name [= value], ...
```

Examples:

```
int x, y, z  
int x = 0, y = 0, z
```

Assigning a tuple is not allowed (this is allowed only with variable already declared)

```
int x, y = 1, 2           ... not valid
```

Node:Constant, Next:[Nil and null](#), Previous:[Declaration](#), Up:[Top](#)

Constant

The constant modifier defines a variable whose value can't change.

Syntax and example:

```
constant type NAME= value

constant int X = 5           ... you can't assign further to x another
value.
constant text T = "demo"
```

Constants declared by users are usually uppercase.

Some predefined constants are keywords of the language:

true	matches an expression that is true.
false	the opposite.
zero	the 0 value.
nil	(Not In List) object that doesn't exist inside a sequence .
null	value of an object while not initialized yet.

PI is a built-in constant and the value is the pi mathematical value.

Node:Nil and null, Next:[Assignment](#), Previous:[Constant](#), Up:[Top](#)

Nil and null

Nil refers to the content of an object, null to its adress.

Null means for "no value" or "declared but not defined", while "nil" means for "not found" or "empty". The null keyword is converted to "null" in PHP and "NULL" in C++. If null is assigned to a variable, the variable can't be referenced, thus is unusable until it is assigned with a value. You can only compare it in a condition with the null keyword.

Nil is not a real value but a construction of the language instead. Here are the values that nil matches for each type...

boolean	false
int	-1
natural	0
real	-1
text	""
array	{}
dict	{}
file	NULL
dir	NULL
object	NULL
dyn	as each of above according to the content.

When nil is assigned to a subscripted interval of an array, this removes the interval from the array.

When Scriptol generates PHP code, nil is replaced by these values:

return of a function:	false
inside an expression:	false
assignment:	as above

Node:Assignment, Next:[Simple assignment](#), Previous:[Nil and null](#), Up:[Top](#)

Assignment

Node:Simple assignment, Next:[Compound assignment](#), Previous:[Assignment](#), Up:[Top](#)

Simple assignment

The syntax of a simple assignment is:

```
identifier = expression
```

You must assign a variable with a value of same type.

```
int i = 10
int j = "343"          ... bad
```

It is possible to convert a type into another with constructors of primitives.

Node:Compound assignment, Next:[Multiple assignments](#), Previous:[Simple assignment](#), Up:[Top](#)

Compound assignment

When a operation performed on a variable, and the result assigned to the same variable, this may simplified in a simple, augmented instruction.

The syntax of such a compound assignment is:

```
identifier compound-operator expression
```

For example, to add 10 to x, and put the result into x, rather than write $x = x + 10$, you can write $x + 10$.

Compound operators are: + - * / mod << >> | & ^

Example:

```
a = 1           ... gives a the value 1
a + 1          ... adds 1 to a
x * y          ... replace the content of x by x * y.
a * (x + 2)    ... a is multiplied by the expression
a = a * (x + 2) ... as above
a & b          ... replace the content of the array a by the intersection of
a and b.
```

Takes not in an expression, $x + 10$ return the addition of x and 10, with no change in x.

```
if x + 1 : ...      x is unchanged.
```

Multiple assignments

A tuple of variables may be assigned a tuple of expressions.

Syntax and example:

```
name [, name]* = expression [, expression]*  
x, y, z = 1, x2, 8
```

The number of expressions at right must match the number of targets at left or may be a single value assigned to several variables.

Multiple assignment allows a function to return several values.

Examples:

```
x, y, z = 0  
a, b = myfunc()
```

Unlike multiple declaration, multiple assignment may be operated on variable having different type.

Example:

```
int x  
text t  
x, t = 1000, "abc"
```

Arrays having a dynamic size, the number of expression to assign is not fixed. Example:

```
x, y, z = array(1, 2, 3, 4)
```

Is as: $x = 1, y = 2, z = 3$.

```
x, y, z = array(1)
```

Is as: $x = 1$

You may assign several variables, previously declared , separated by commas from:

- a value or expression.
- an array.
- values of a dict (use the values() method).
- a call of function.
- a tuple of values or expressions separated by commas.

In case of array or dict, or a tuple, or a function returning several values, the variables from 1 to n are assigned the items from 1 to n, in the same order.

If the number doesn't match, it is an error.

If the function returns a single value, the same value is assigned to each variable at left.

Conditional assignment

This special assignment is intended to change a property or other variable that has already a default value, when a new value is given.

The := symbol assigns conditionally a variable providing the value to assign is not evaluated as nil.

The error flag is set to "true" if the expression to assign is nil.

Example:

```
x := z
error ? print "z is nil"
```

The above assignment is equivalent to:

```
if z <> nil
  x = z
else
  print "z is nil"
/if
```

Node:Operators, Next:[Expression](#), Previous:[Conditional assignment](#), Up:[Top](#)

Operators

Comparison operators are:

```
=      (equal)
<      (less than)
>      (greater than)
<=     (less or equal)
>=     (more or equal)
<>     (not equal)
```

The "in" operator tests the inclusion of an element in a sequence: a string in a text, an object in an array, a value in a range.

```
if "a" in line print "in text"
if x in 1..10 print "in range"
```

Binary operators are:

```
&      (and)
|      (or)
^      (exclusive or)
~      (not)
<<     (shift left)
>>     (shift right).
```

Array operators are:

```
&      (intersection)
|      (union)
^      (complement of intersection).
```

Precedence

Unary operators have precedence over binary ones. Among binary operators, precedence must be denoted by parenthesis.

Node: [Expression](#), Next: [Precedence](#), Previous: [Operators](#), Up: [Top](#)

Expression

An expression is a combination of values and operators. Main kinds of expressions are:

- **Arithmetical expressions:** Combinations of arithmetical values or function calls with these operators: +
- * / mod.

mod is a shortcut for modulo and returns the remainder of a division...

Example:

```
print 100 mod 3
... should display 1 that is the remainder of 100 divided by 3.
```

- **Conditional expression:** Set of two at least expressions linked by relational operators, and returning a boolean value.

```
=    equal
<    less
>    greater
<=   less or equal
>=   greater equal
<>   different (the operator != is valid also)
```

- **Logical expressions:** A combination of boolean values or expressions (relational or logical ones) and logical operators and, or, not.

The "and" expression returns true if the two terms are true, false otherwise.

The "or" expression returns true if one of the two terms is true, false if the two ones are false.

If we add the "not" operator on the whole expression, it negates the final result.

Example:

```
boolean x = true
boolean y = false
if x and y print "true"; else print "false"           ... should display false
not(x or y)           ... is false.
x and not y           ... is true
```

- **Binary expressions:** A set of numbers linked by binary operators.

```
|    binary or
&    binary and
^    exclusive or
~    binary not
<<   shift left, that is equivalent to multiply by 2
>>   shift right, that is equivalent to divide by 2
```

- **Text expressions:** Operators on text are:

```
= < > <= >=   to compare two texts.
+           to concatenate two texts.
[]         indexing, slicing or splicing (see at text chapter).
```

in test if a text is a part or another one.

Example:

```
text t = "prefix"
print t + "suffix"
...should display: prefixsuffix
```

- List expressions: Operators on lists (array, dict) are:

```
= < > <= >= <> compare two lists for values.
+ concatenates two lists (doubles may result).
- removes from a list, element of a second one (but doubles).
[] indexing, slicing or splicing (see at array and dict).
in tests if an element is inside a list.
& intersection, returns element common to two lists.
| union, returns elements of two list without doubles.
^ complement of intersection.
```

Node:Precedence, Next:[Function](#), Previous:[Expression](#), Up:[Top](#)

Precedence

Some programming languages have instituted precedence rules, so, when parenthesis are missing, we know the terms involved by each operator. Meanwhile precedence has been built inside the Scriptol parser, error message is thrown when parenthesis are missing because they are required for readability. The sole case where precedence is admitted without parenthesis is for unary operators: not, ~
Unary operator applies always to the term that follows it, and thus to be applied to an expression, the expression must be enclosed between parenthesis.

Examples:

```
if not a and b
if not (a and b)
```

In the first case, the not operator is associated to the "a" variable.
The second line negates the whole expression in parenthesis.

In compound assignment the first operator, that is also a equal operator, applies to the whole expression at right.

Node:Function, Next:[Alias](#), Previous:[Precedence](#), Up:[Top](#)

Function

A function starts with a header and ends with the "return" keyword.
The return type is required, and the type of arguments also. "void" is used if nothing is returned.

Syntax:

```
type [,type]* identifier( [argument [, argument]*] )
... statements ...
```

```
return [expression [, expression]*]

int myfunc(int x)

int, boolean myfunc(text t, int i)
```

Example:

```
int multiply(int x, int y)
    int z
    z = x * y
return z
```

This may be written simply:

```
int multiply(int x, int y)
return x * y
```

The body is a list of statements, including some "return" if needed. Inside the body of a function you can put any kind of statement, but a high-level declaration (class, enum) or another function. You can put inner return statements.

The ending statement is a return with zero, one, or several values.

Examples:

```
return
return x
return x, y, 5
```

If the function return several values, it must have also several return types and the return statement has several parameters (in the same order that the return types).

Example:

```
text, int, int coordinate(int num)
    int x = mytable[num].x
    int y = mytable[num].y
return "coo", x, y
```

Calling a function:

A call to a function may assign zero, one, or several variables.

```
myfunc()
a = myfunc()
a,b,c = myfunc()
```

Node:Alias, Next:[Default values](#), Previous:[Function](#), Up:[Top](#)

Alias: Arguments of a function

When a function has object as arguments, the name of the arguments are aliases of the original objects, and any change inside the function are made in fact on the original objects.

By default, primitives are copy and other arguments are aliases. It is possible, if required, to use also the original of a variable, rather than a copy: thanks to the "alias" keyword, the compiler knows that this is

just another name for the original variable.

Example:

```
void func(number x)
  x + 1
  print x
return
```

```
void funcalias(alias number x)
  x + 1
  print x
return
```

```
number y = 5
func(y)
print y
... should display 6 then 5
```

```
funcalias(y)
print y
... should display 6 then 6.
```

Node:Default values, Next:[Scope and function](#), Previous:[Alias](#), Up:[Top](#)

Default values

Assigning a default value allows to omit an argument at the function call. The complete syntax for the heading is:

```
type [,type]* name(type name [= expression] [, type name [= expression]]* )
```

Example:

```
int increment(int x, int y = 1)
  x + y
return x
```

```
print increment(10, 5)           ... should display: 15
print increment(10)              ... should display: 11
```

The default value 1 has been used to replace the missing parameter.

If the interface of a function has several arguments with a default value, you can't omit one in the call without omitting all following ones. The arguments can't be not recognized by their type.

If what you want if to write a function with different numbers and types of parameters at call, you must use an array or a dict instead.

Example:

```
void param(dict parlist)
  size = parlist["size"]
  name = parlist["name"]
  values = parlist["values"]
return
```

In this example, the variables "size", "name", "values" are global, and they are just assigned by the content of the dict in argument. For this example to be perfect in fact, we should use conditional assignments. This may be achieved thanks to the conditional assignment symbol :=

`x := y`

x is assigned the value of y only if y is not nil.

Node:Scope and function, Next:[Quiet mode](#), Previous:[Default values](#), Up:[Top](#)

Scope and function

A function opens a new scope for all variables declared inside. If the function is at the global level, all global variables compiled before the function are visible in the function.

Inside a function, a variable can't be declared with same name that a global variable. This rule applies also inside methods of classes.

If the function is a method of a class, all variables declared in the class before the function are visible in the function.

Objects declared in the function are visible in embedded control structures. Identifiers still in use can't be reused inside embedded blocks too.

Node:Quiet mode, Next:[Main](#), Previous:[Scope and function](#), Up:[Top](#)

The quiet mode

It is possible to avoid PHP display errors and warning if the name of the function in calls is prefixed by the @ symbol.

This is ignored by the binary compiler.

```
@myfunct(x, y)
```

Node:Main, Next:[Print and echo](#), Previous:[Quiet mode](#), Up:[Top](#)

The "main" function

It is often required to pass arguments to a program from the command line. To do that in Scriptol (as in PHP), use the external \$argv PHP variable.

To mimic the way C++ passes arguments to a program, declare a function "main", and call it with \$argv (and \$argc if needed) as argument.

```
int main(int argnum, array arglist)
    print argnum, "arguments"
    scan arglist
        print arglist[]
    /scan
return 0
```

```
main($argc, $argv)    ` argc and argv are external system variables.
```

The main function may be declared either as above or without argument:

```
int main()
int main(int, array)
```

Print and echo

Echo

Echo displays an expression, or a list of expressions separated by commas, without blank space nor line feed at end (as the print statement does).

Syntax: echo expression [, expression]

Example:

```
x = 5
y = 20
z = 1
echo "values", x, y / 2
echo z
```

This will be displayed on the screen as this:

values5101

Example:

```
echo "demo", 5
echo "next"
```

displays: demo5next

Print

In a such case, a better display is performed with the print statement. Print replace the comma by a blank space, and it adds a newline at end.

Syntax: print expression [, expression]

Example:

```
print "demo", 5
```

displays: demo 5

A single print statement, without argument, sends a line feed.

Example:

```
print
```

Input

To enter a text, or numbers, from the keyboard, use the input command.
A text may be displayed before the input is requested.

Example:

```
text name
input name
print name
```

Example:

```
input "who are you? ", name
```

The variable to assign must be declared before the input command, it may be a text or any kind of number.

Node:Control structures, Next:[lf](#), Previous:[Input](#), Up:[Top](#)

Control structures

Control structures are:

```
if
  if else /if.
  one-instruction if.
  composite if.
for
  for /for.
  one-instruction for.
  options: in interval, in array.
scan
  by function.
  one-instruction scan.
  scan /scan.
while
  while let.
  while /while.
  while forever.
do
  do until.
  do /do condition.
  do case / else / always /do.
enum
  simple enum.
  dict enum.
error
  error /error.
  one-instruction error.
```

break and continue are two statements used inside controls structures.

Scriptol has different syntaxes for short control structures in a single instruction, or complex one with a block of instructions.

A one-instruction control structure has the form:

```
keyword expression let statement
or keyword expression ? statement
or keyword expression statement-starting-with-a-keyword.
```

"Let" means for "Last statement, Execute and Terminate". The let element is always the last part of a control structure and may be the sole one. In this case there is no ending but the end of line. The sole statement may be any basic statement and can't be a control structure.

Let is required when the statement is an assignment or a function call, but is optional when a keyword follows it.

The "?" symbol is just a shorter replacement for "let".

A multi-lines control structure has the form:

```
structure-name expression [:]
... statements ...
/structure-name
```

The colon is optional (but if a statement continues the ligne).

Example with separator:

Example:

```
if a = b : print t ; /if
```

Node:If, Next:[Composite if](#), Previous:[Control structures](#), Up:[Top](#)

If

One-instruction syntax:

```
if boolean-expression let statement
[else statement]
or:
if condition let statement ; else statement
```

One must read the line as this: is condition true? if yes, action...

The keyword let or the symbol ? are equivalent. The separator is : or end of line when a bloc of instructions follows.

Examples:

```
if a = 5 ? break
if a < 5 ? print "less"
else print "more/equal"
if a = 1 ? print "one"; else print "several"
```

Multi-lines syntax:

```
if boolean-expression [:]
... statements ...
else ...optional
... statements ...
/if
```

N.B.: The colon is optional after the condition, as the semi-colon after a statement, but is required to

concatenate the lines.

Examples:

```
if (x + y) > 8
  print "> 8"
  print x
else
  print "<= 8"
/if
```

```
if x + y) > 8 : print "> 8" ; print x ; else print "<= 8"; /if
```

Node:Composite if, Next:[For in](#), Previous:[If](#), Up:[Top](#)

Composite if

The switch case control structure of C++ or Java is not implemented in Scriptol because it is too weak and useless (in Scriptol).

It is replaced by a more powerful variant of the if construct, that can match any type of variable, and various types of comparisons.

The syntax is:

```
if not-boolean-expression [:]
  operator expression : statements
  operator expression : statements
  ...
else
  ... statements ...
/if
```

A not-boolean expression, is an ident, a litteral, or any expression that doesn't return a boolean value (true or false).

The structure: "operator expression : statements" is a case group. There are no "break" keyword after a case group. A break will causes exiting from the control structure that would contain the current "if" structure.

Valid operators are:

=, <, >, <=, >=, !=, <>, in, else.

"else" here is equivalent to "default" in the switch case of C.

Examples:

```
if a
= 1: print 1
= 2: print 2
> 2: print ">2"
else
  print "<1"
/if
```

```
if x
  in array(1,2,3): print "first"
  in array(4,5,6): print "second"
  in array(7,8) : print "third"
  else print "other"
/if
```

For in

The for control structure scans either a range or a sequence and puts each item into a variable.

Syntax of for range:

```
for variable in start..end [step s] [:]
  ... statements ...
/for
```

The start, end, step elements are either identifiers or literals.

- step is optional, the default value is 1.
 - the end value is included in the range if the range symbol is "..", not if the symbol is "-".
 - the end value may be inferior to the start one, providing the step is present and holds a negative value.
- The container variable that is assigned each value of the range, may be declared into the heading, in this case, it is local to the control structure.

Syntax of for list:

```
for variable in list-expression [:]
  ... statements ...
/for
```

In this case, the content of an expression is scanned and each item assigned to the variable. The expression must be an array, a text or any expression that returns an array or a text.

Examples:

```
for i in 1..10
  print i
/for

for w in myarray
  print w
/for
for w in arr1 + (arr2 & arr3[0..5])
  print w
/for
```

The one-instruction shortened syntax is:

```
for ident in list let basic-statement
```

Examples:

```
for w in mylist let print w
for x in 10..1 step -1 let print w + 10
```

Scan by

The scan structure allows to scan an array and apply a function onto each element of the array.

Syntax of scan:

```
    scan a  by f
or
    scan a [,b, etc...]
        ... statements ...
    /scan
```

- a, b, etc... are arrays.

- f is the name of a function.

The scan by format supports only one array for compatibility with PHP 5, the tag format support several arrays. A function may be called from the body.

This applies the function to each item of the array.

For the function to modify the array, the "alias" modifier must be put before the type of the argument in the declaration of the function. Scan by requires a user function, and does not work with method of class.

In the block format, each item of the array(s) is(are) processed by the statements inside the body of the control structure. The currently pointed out item, with unknown index, is accessed with the [] empty indexing.

The one-instruction syntax is also allowed:

```
scan a let statement
```

Example:

```
array a = {1,2,3,4 }
void fun(number x) : print x * x; return
scan a by fun

scan a : print a[] * a[]; /scan
```

These two examples have the same result.

One can modify the array by an assignment as: a[] = ...

Example:

```
scan a let a[] = 0

scan a
    a[] * a[]
    print a[]
/scan
```

Example with several arrays:

```
void mulfun(dyn a, dyn b) print a * b; return
scan a,b by mulfun
or:
scan a,b let print a[] * b[]
```

Scanning a two-dimensional array

Here is an example to create and scan a two-dimensional array:

```
array a = ( ("a", "b", "c"),
            ("x", "y", "z"),
            (1, 2, 3))
```

This array really holds three dyn (dynamic variables) that hold an array each one.

In this example, the element of a, that is a dyn, is casted to array, since it holds an array and we want this array assigned to x. Without the cast, the element will be inserted into the x array.

```
scan a
  array x = a[].toArray()
  scan x
    print x[]
  /scan
/scan
```

It would be simpler to avoid creating the xx temporary array, this may be done as in this example (using the one-statement syntax):

```
scan aa
  scan aa[] print aa[][]
/scan
```

This shortened form works with the interpreter and the native compiler, not with PHP (version 4.2).

Node:While, Next:[While let](#), Previous:[Scanning a two-dimensional array](#), Up:[Top](#)

While

The while structure is a conditional loop.

One-instruction syntax:

```
while expression let instruction
```

Standard syntaxe:

```
while expression [:]
  ... statements ...
/while
```

Example:

```
int x = 10
while x < 20
  print x
  x + 1
/while
```

A "break" statement exits the loop.

A "continue" statement skips all that follows and starts a new loop.

Node:While let, Next:[Do until](#), Previous:[While](#), Up:[Top](#)

While let

This syntax is recommended to avoid the risk of infinite loops.

The statement that changes the value of the expression that is the condition of the loop, is moved after the /while marker by the mean of the "let" statement.

Example:

```
while x < 10
  if (x mod 2) = 0 continue
  print x
/while let x + 1
```

The continue statement jump to the let instruction.

There is not corresponding code in C or PHP, because a "continue" statement in C or PHP bypasses following instructions, including "x + 1", and leads to a infinite loop.

Simplified syntax of while let:

```
while condition
  ...statements...
let incrementing
```

Example of one-statement loops:

```
while x < 10 let x + 1
while x < 10 : print x; let x + 1
```

The condition of the while loop may be the "forever" keyword, and we enter an infinite loop, exited by a "break".

Node:Do until, Next:[Do case](#), Previous:[While let](#), Up:[Top](#)

Do until

The block of statements enclosed inside the do /do tags, is a new scope, and enclosed statements are performed first, before a condition is tested, if a condition is given.

General syntax of do:

```
do
  ... statements ...
/do [ condition ]
```

And "/do condition" may be shortened in "until".

The block of statements inside do until is performed while the condition is false, and is exited when the condition is true.

Syntax of do until:

```
do
    ... statements ...
until expression

do
    print x
    x + 1
until x = 10
```

Node:Do case, Next:[Do extended syntax](#), Previous:[Do until](#), Up:[Top](#)

Do case

Do case is a powerful pattern-matching control structure. It contains one or several case groups followed by an optional else and an optional always. One case group only is processed, the one the condition of which if first matched. Always is always executed. Else only when no condition is matched.

Syntax of do case:

```
do
    case condition : statements
    [ case condition : statements ]
    [ else statements ]
    [ always statements ]
/do [while expression]
```

- A condition if followed by a colon or a newline. After the "else" and "always" keywords optionally.
- Only one case is performed, or none if no expression is matched.
- The "else" group is equivalent to "default" in the switch case of C.
- The "always" group, if present, is performed in any cases.
- May end with /do, /do forever, /do while expression.
- When the forever or the while option are used, the construct becomes a DFA (Deterministic Finite-state Automata). A break may be required to exit it.
- A "break" must not be used at end of a case group as in C: it will exit the whole construct, not the case group!

Examples:

```
do
case nom = "pussycat":    print "it is a cat"
case nom = "flipper":    print "it is a dolphin"
case nom = "mobby dick": print "it is a whales"
else
    print "another animal"
/do

int state = ON
do
    case state = ON: counter + 1
    case state = OFF: break
    always
        state = getState() ` some function
/do while forever
```

The automaton loops until the OFF state is encountered (produced by the called function).

Node:Do extended syntax, Next:[Break and continue](#), Previous:[Do case](#), Up:[Top](#)

Do extended syntax

The /do ender may be completed by a condition, while or forever.
Statements while be processes once, and while the condition is true.

Syntax of do while:

```
do
    ... statements ...
/do while boolean-expression

do
    print x
    x + 1
/do while x < 3
```

Options

- do ... /do ... inner bloc of statements, with case groups or no.
 - do ... /do while forever ... infinite loop
 - do ... /do forever ... infinite loop in a shortened form
- infinite loop requires a break to exit.

Node:Break and continue, Next:[Enum](#), Previous:[Do extended syntax](#), Up:[Top](#)

Break and continue

Break is the command to exit a loop.

Example using the "forever" keyword that leads deliberately into an infinite loop:

```
int x = 0
while forever
    print x
    if x > 100
        break
    /if
    x + 1
/while
```

When x reaches the 100 value, the break statement skip instructions beyond /while, and the while structure.

Continue is a command to skip all statements, from the current position, up to the end of the structure, and thus to start a new loop.

```
int x = -1
while x < 20
    x + 1
    if (x mod 2) = 0 continue
    print x
/while
```

This example displays only the even values of x, because if an odd value is encountered, the condition `x mod 2` is matched and a `continue` command is performed.

Note that if incrementing the variable tested in the `while` condition is put after the `continue` statement, it is skipped also. An infinite loop is prevented thanks to the `while ... let` syntax.

Node:Enum, Next:[Indexing](#), Previous:[Break and continue](#), Up:[Top](#)

Enum

Enum allows to assign sequentially values of several types to identifiers.

You can use also equal to assign an integer, for the automatically generated sequence continue, starting from this number.

You can also assign to any identifier, a real or a text, with a colon.

Syntax of enum:

```
enum
  identifier [ : value | = value ]
  [ identifier ...]*
/enum
```

```
enum
  Z,
  ONE,
  TWO,
  THREE
/enum
```

This assigns 0 to Z, 1 to ONE, and so on and is equivalent to:

```
constant int ZERO = 0
constant int ONE = 1
etc...
```

or:

```
enum: Z:0, ONE:1, TWO:2 /enum
```

More complex example assigning various values and restarting a sequence:

```
enum
  Z : "a0",           ... assign a0
  ONE : "a1",         ... assign a1
  THREE,              ... assign 0
  FOUR : 3.15,        ... assign 3.15
  FIVE = 5,           ... assign 5 and restart the numbering
  SIX                 ... assign 6
/enum
```

Example of the one-line syntax:

```
enum ZERO, ONE, TWO, THREE
```

Node:Indexing, Next:[Range](#), Previous:[Enum](#), Up:[Top](#)

Indexing

The syntax of an index in a text or array is: [indice].

The indice must be a simple expression without square brackets embedded inside. A simple expression is a literal number, an identifier, a function call, or an arithmetical expression, and must be resolved as an integer value.

Node:Range, Next:[Subscripting](#), Previous:[Indexing](#), Up:[Top](#)

Range

The syntax of a range is:

```
start .. end
```

It is enclosed between square bracket to subscript a list:

```
list-name[start .. end]
```

Start and end are integer expressions.

The end is part of the range unless the "-" operator is used instead.

```
a[0 -- 100]      is as a[0 .. 99]
a[x -- y]       is as a[x .. y-1]
```

Examples of ranges:

```
0..10
x..y
x * 2 / 4 .. y + 10
```

You may use range to:

- test if a value is inside a range: if x in 10..100
 - scan a range: for x in 1..100
 - extract a part of a list: array b = a[x..y]
 - change a part of a list: a[x..y] = another-list
-

Node:Subscripting, Next:[Text](#), Previous:[Range](#), Up:[Top](#)

Subscripting

The start and end value may be omitted when subscripting a list.

Splitting a list, there are three ways:

```
array a = array(x0, x1, x2, x3, x4, x5, x6, x7, x8)
array b = a[..2]
array c = a[3..5]
array d = a[6..]
```

Now we have three arrays, with these contents:

```
b: (x0, x1, x2)
c: (x3, x4, x5)
d: (x6, x7, x8)
```

Displaying the arrays:

```
b.display()
c.display()
d.display()
```

You should see:

```
array (
  [0] => x0
  [1] => x1
  [2] => x2
)
```

and so one...

To replace an interval by another list, a simple assignment is required:

```
a[3..5] = array("a", "b", "c")
```

The content becomes:

```
(x0, x1, x2, "a", "b", "c", x6, x7, x8)
```

With the same syntax, a sub-list may be replaced either by another list, or by a single value:

```
a[3..5] = "xyz"
```

The original list becomes:

```
(x0, x1, x2, "xyz", x6, x7, x8)
```

If we want rather to remove a sub-list from the original array, we have to declare it "not in list":

```
a[3..5] = nil
```

As we have removed the sub-list 3..5 that is (x3, x4, x5), the content now becomes:

```
(x0, x1, x2, x6, x7, x8)
```

We can test is a valuer is in a sub-list.

Example:

```
array a = array("one", "two", "three")
if "two" in a[2..5] print "inside"
```

Node:Text, Next:[Methods on text](#), Previous:[Subscripting](#), Up:[Top](#)

Text

A text is a basic objet with methods, that holds a string of characters. A literal text is a string enclosed between simple or double quotes. When a text is the argument of a function, the function uses a copy of the text, not an alias on the original one.

An indice may be negative in slicing or splicing, not in indexing.

Syntax:

text s	creates a text.
s = "str"	initializes.
s = s[i]	gets a char.
s[i] = s2	replaces a char, s2 should be a one-char text.
s = s[i..j]	gets a sub-string, from i until j included.
s[i..j] = s	replaces a sub string.
s[i..j] = ""	removes a sub string.

A literal text is a string of character enclosed in simple or double quotes.

The + symbol may be used with texts as with mathematical expression to concatenate strings.

Example:

```
text b = "prefix"  
text a = b + "suffix"
```

The content of a will be: "prefixsuffix". T

Node:Methods on text, Next:[Dynamic variables](#), Previous:[Text](#), Up:[Top](#)

Methods on text

Return	Method	Function
void	cat(text)	concatenates another text.
text	capitalize()	converts the first char to uppercase.
int	compare(text)	compares lexicographically two texts (ignore case). returns -1, 0, 1.
text	dup(int)	returns a text duplicated n times. Ex: "*"dup(10).
void	fill(text, int)	fills the text with the text argument n times.
int	find(text t2)	returns the position of text s2 inside the text. returns "nil" if no found. (test for x = nil as x <> nil doesn't work in PHP)
int	identical(text)	compares, doesn't ignore case. Return -1, 0, 1
void	insert(int,text)	inserts a text at position.
boolean	isNumber()	return true if the test is a numeric string.
int	len()	returns the length.
int	length()	returns the length.
text	lower()	converts to lowercase.
text	lTrim()	removes heading controls/blanks.
void	replace(ft, rt)	replaces each occurrence of ft by rt.
void	reserve(int)	allocates the size to use the text as a buffer.
text	rTrim()	removes trailing controls/blanks.
array	split(sep)	splits a text into items separated by sep.
int	toInt()	converts to integer.
natural	toNatural()	converts to natural.
real	toReal()	converts to real.
text	toText()	converts a literal string to text (for C++).
text	trim()	removes heading and trailing control codes/blanks.
text	upper()	converts to uppercase.
text	wrap(int size)	wordwraps the text.

Node:Dynamic variables, Next:[Sequence and list](#), Previous:[Methods on text](#), Up:[Top](#)

Dynamic variables

Are declared with the type "dyn".

Dynamic variables have the methods of all other types of primitives but not the methods of declared classes (see at "extern").

Once a variable is assigned to a dyn, a cast is required to assign the content of the dyn to a typed variable:

Methods on dyn:

- cast methods: toBoolean, toInt, toInteger, toReal, toNumber, toNatural, toText, toArray, toDict, toFile, toObject.
 - test methods: isBoolean, isInt, isInteger, isReal, isNumber, isNatural, isText, isArray, isDict, isFile, isObject.
 - arrayType method (return the type of a typed array).
-

Node:Sequence and list, Next:[Array](#), Previous:[Dynamic variables](#), Up:[Top](#)

Sequence and list

A sequence is either a static (text) or dynamic, associative list (array or dict).

List and sequence have same operators, but & and | that are proper to lists.

Operations on lists are these:

```
[] : index, slice, or splice.
+  : merges two sequences. Or push a scalar to a dynamic list.
-  : removes a sequence from another one, or an item from a dynamic list.
=  : compares two sequences.
in : tests if an object is in a sequence.
&  : intersects two lists (gives common elements).
|  : union without doublings of two lists.
```

Node:Array, Next:[Making an array](#), Previous:[Sequence and list](#), Up:[Top](#)

Array

Two types of dynamic lists of objects exist in Scriptol:

```
array: keys are integer numbers.
dict:  (dictionary) keys are texts.
```

An array is a dynamic and indexed list of object or literals.

Dynamic means the size is not predefined nor limited.

An empty array is symbolized by {}.

A literal array is a list of expressions separated by commas and enclosed between curly braces. A variable is a valid element.

The array constructor starts with the keyword "array".

Type a.display() to view the content of the a array, this output is displayed:

```
array(
  0 : first
  1 : second
  2 : last
)
```

Node:Making an array, Next:[Indexing an array](#), Previous:[Array](#), Up:[Top](#)

Making an array

The array constructor has the form: `array(value, value, etc...)`

An empty initializer is written: `array()`

A literal array is written: `{ ... values ... }` You can define an array by assigning a literal array or the constructor, or a single value.

Syntax:

```
array a                                creates an array.
array a = array()                       creates an empty array
array a = {}                            creates an empty array
array a = array(x, y, ...)               creates and initializes an array.
array a = {x, y, ...}                   creates and initializes an array.

array a = { 8, 9 }                       assign a literal array.
array a = array(8)                       using the constructor.
array a = 3                              create an array of one element.
```

Elements of an array may be any expression ******* but boolean ones *******. If you put the true value inside an array PHP will return always true when you use the "in" operator, with any searched value.

An array may be declared without assigning any content, and filled further:

```
array a
a.push("first")
a.push("second")
a.push("third")
```

Elements are accessed by their position inside the array. Examples:

```
a[1] = "a"
a[2] = "b"
scan a print a[]
... should print: a b
```

Node: [Indexing an array](#), Next: [Iterator](#), Previous: [Making an array](#), Up: [Top](#)

Indexing an array

Items inside an array are accessed by an integer number.

Syntax:

```
a[n]                                    reads the item with at position n.
a[n] = x                                replaces the item at n by x.
a[n] = nil                               erases the item at n
a = {}                                   clears the whole array.
a[n].upper()                             calls a method on the dynamic element n.
```

An empty index designate the current item:

```
a[]                                    reads the item with at current position.
```

The indices may be any kind of expression, but this expression must not include another array.

```
a[10 + x / 2]                          valid.
```

```
a[10 + b[5]]          not valid.
```

Since arrays may contains any kind of object, and even other arrays, you need for a dynamic variable to get an element unless you know the type of the item to get:

```
array a = { x, "two", 3 }
dyn x = a[1]           use dyn for unknow element at first position
text t = a[2]
int i = a[3]
```

When an element is added outside bound, it is just pushed at last position into the array.
If you assign an element with the statement:

```
a[1000] = "x"
```

En PHP:

```
array(
  0 : one
  1 : two
  2 : last one
  1000: x
)
```

En C++:

```
array(
  0 : one
  1 : two
  2 : last one
  3: x
)
```

In PHP, the 1000 indice is just temporary and will differ as soon that a statement has modified the array. For compatibility issue, use a such statement to replace values in an array, not to add them, this is the way dict only may be filled.

Node:Iterator, Next:[Using array as a stack](#), Previous:[Indexing an array](#), Up:[Top](#)

Iterator

An array may be scanned with an iterator.

One points out the first element with `begin()`, or the last one with `end()`. The currently pointed out value is accessed by an empty indice: `[]`.

Iterator methods:

```
begin()    points out the first element and returns it.
end()      points out the last element and returns it.
inc()      moves to the next element.
           Returns the value, or nil, beyond the last element.
dec()      moves to the previous element. Returns the value or nil.
index()    returns the index of the pointed out element.
value()    returns the value of the pointed out element.
[]         empty indice for the current element.
nil        means for "not in list" and is returned by various functions
           when no value can be returned.
```

Example of use with the a array:

```
a.begin()    ` moves to the first element
```

```
while not (a[] = nil)      ` tests if end of list reached
    print a[]             ` prints the currently pointed out element
    a.inc()               ` moves to next element
/while
```

In reverse order:

```
a.end()
while not (a[] = nil)
    print a[]
    a.dec()
/while
```

Node:Using array as a stack, Next:[Interval in array](#), Previous:[Iterator](#), Up:[Top](#)

Using array as a stack

Once the array is created, you can perform various list - or stack - processing...

```
a.push("item")           ...add an element at end of the list
a.unshift("item")       ...insert an element at begin of the list
a.pop()                 ...read and remove the last element
a.shift()               ...read and remove the first element
```

You can both read and remove the elements of an array by successive such statements:
print a.shift()

Node:Interval in array, Next:[Operators on array](#), Previous:[Using array as a stack](#), Up:[Top](#)

Interval in array

An interval is subscripted by a couple of positions.

```
a[pos..end]             the range between "pos" and "end" included.
a[..end]                from the start to position "end".
a[pos..]               from position "pos" to the end of array.
a[..]                  gets the whole array (useless)
a[pos] = nil           removes an item (keys are renumbered).
a[pos..end] = nil     removes a range of items (here also).
a[pos..end]= b        replaces a range by an array/item
```

Node:Operators on array, Next:[Multi-dimensional array](#), Previous:[Interval in array](#), Up:[Top](#)

Operators on array

An item, or a group of items, may be added or removed with the + and - operators.

Example:

```
array a = array("one", "two")
```

```

array b
b = a + array("three", "four")
"three", "four").
b = b - array("one", "four")

```

b is now ("one", "two",
b is now ("two", "three").

Only the + and - arithmetical operators are usable on arrays along with the in operator.

The "in" operator

This operator may be used to test if a value is contained inside a list (array, dict or even a text), and to scan the content too.

Syntax and examples::

```

if variable in array ... some statement ...
for variable in array ... some statement...

if x in a print "inside"
if x in array(1,2,3) print "inside"

for x in a print x
for t in array("one", "two", "three") print t

```

Binary operators on dynamic lists

You may use for dynamic lists (array or dict) the binary operators:

```

& intersection    returns only elements that are parts of both the two lists
| union           merge two list without doubloons.
^ complement of intersection

a = array(1,2,3,4) & array(3,4,5,6)  assigns (3, 4) to a.
a = array(1,2,3,4) | array(3,4,5,6)  assigns (1,2,3,4,5,6) to a.

```

Node:Multi-dimensional array, Next:[Content of an array](#), Previous:[Operators on array](#), Up:[Top](#)

Multi-dimensional array

The number of dimensions is not limited.

For a two-dimensional array, the syntax...

- to access an element is: x = arrayname[i][j]
- to change an element is: arrayname[i][j] = x

To create an element, the syntax is:

```

arrayname[i] = array()
arrayname[i][j] = x
or
arrayname[] = array(x)

```

You cannot create directly an element into a not existing sub-array. The i and j indexes suppose that i and j elements already exist.

Node:Content of an array, Next:[Dictionary](#), Previous:[Multi-dimensional array](#), Up:[Top](#)

Content of PHP array, step by step

We have to know how integer keys of PHP array change, according to all operations we can perform. This is important to understand associative arrays and to avoid lot of bugs. After each operation, the content is displayed.

```
array a = {}  
array  
(  
)
```

```
a.push("one")  
array  
(  
  [0]=> one  
)
```

```
a + array("two", "three", "four")  
array  
(  
  [0]=> one  
  [1]=> two  
  [2]=> three  
  [3]=> four  
)
```

```
a.shift()  
renumbered.
```

The first element is removed and the keys are

```
array  
(  
  [0]=> two  
  [1]=> three  
  [2]=> four  
)
```

```
a[1000] = "thousand"  
array  
(  
  [0]=> two  
  [1]=> three  
  [2]=> four  
  [1000]=> thousand  
)
```

```
a.unshift("x")
```

All keys are renumbered, even the 1000 one.

```
array  
(  
  [0]=> x  
  [1]=> two  
  [2]=> three  
  [3]=> four  
  [4]=> thousand  
)
```

Creating two new arrays:

```
array a = ("one","two")  
array b = {}  
b[1000] = "thousand"  
a + b
```

Keys are renumbered.

```
array  
(  
  [0]=> one  
  [1]=> two  
)
```

```
[2]=> thousand
)
```

If we replace `a + b` by `a.push("thousand")` the result will be the same.

Node:Dictionary, Next:[Making a dict](#), Previous:[Content of an array](#), Up:[Top](#)

Dictionary

A dict is a dynamic list of pairs key and value. Keys are always texts. Values may be any objects. Key and value may be variables.

The format for a pair key and value is: `key:value`.

(The PHP equivalent is `key => value`).

An empty dict is symbolized by `{}`.

A literal dict is a list of pairs separated by commas and enclosed in curly braces.

Unlike arrays, a dict is filled by assignments:

```
d[k] = "b"
d["first"] = "element 1"
```

Array and dict share same methods, but some ones are more relevant with arrays, other ones with dicts.

Node:Making a dict, Next:[Indexing a dict](#), Previous:[Dictionary](#), Up:[Top](#)

Making a dict

A dict may be assigned a constructor or a literal dict.

Syntax:

```
dict d                                creates a dict.
dict d = {x:v, y:w, ...}              creates and initializes a dict.
dict d = dict(x:v, y:w, ...)         creates and initializes a dict.
```

The values stored may be any kind of objects.

The key can be a variable and the value an expression.

Examples:

```
text k = "a"
text v = "b"
dict d = dict(k : v)
dict d = dict(k : v + "x".dup(4) + 20.toText())
```

This example puts `"bxxxx20"` into `d` with the key `"a"`.

Node:Indexing a dict, Next:[Interval and dictionary](#), Previous:[Making a dict](#), Up:[Top](#)

Indexing a dict

Items in an dict are accessed by a textual key.

Syntax:

```
d["key"]           gets the first item with the key "key".
d[key] = dyn       replaces a value or add the couple key, value
                   if the key is not already inside the dict.
d[key] = nil       removes an item.
d = {}             clears the whole dict.
```

Example:

```
dict d
d["program"] = "what we want to write faster"
print d["program"]    ... will display the text above
```

Node:Interval and dictionary, Next:[Methods of array and dict](#), Previous:[Indexing a dict](#), Up:[Top](#)

Interval and dictionary

The right way to use a dictionary is by the means of keys or iterator. In some cases, it may be useful to access a range of items directly.

When adding an element or another dictionary to a dict, by the way of interval, push, unshift, PHP generate a new key for the item. The new key is a number.

- If you replace a range by another dict, some of the items may be lost.
- This also does happen when merging.
- The keys of the replacing dict are not kept in changed dict.

Examples of display:

Should print all keys and values in the dictionary.

```
dict d = {"a":"alia", "b":"beatrix", "c":"claudia"}           creating the
dictionary

d.display()                                                   displaying

for k,v in d : print k,v; /for                                 displaying by a for loop

number i = 0
d.begin()
while i < d.size()                                           displaying by an iterator
  print "$i", d.key(), d[]
  d.inc()
/while let i + 1
```

Example: get the last couple

```
v,k = d.end(), d.key()
```

the order is important as d.end() move the pointer to the last element

Node:Methods of array and dict, Next:[Typed arrays](#), Previous:[Interval and dictionary](#), Up:[Top](#)

Methods of array and dict

Return	Name	Action
dyn	<code>begin()</code>	points out the first item.
dyn	<code>dec()</code>	decrements the pointer and returns the element.
void	<code>display()</code>	displays the array.
dyn	<code>end()</code>	points out the last item.
boolean	<code>empty()</code>	returns true if array empty.
int	<code>find(dyn)</code>	searches for an item, returns the index or nil.
void	<code>flip()</code>	values become keys and conversely.
dyn	<code>inc()</code>	increments the pointer and returns the element.
int	<code>index()</code>	returns the index of pointed out item.
void	<code>insert(int, dyn)</code>	inserts a value at the integer position.
text	<code>join(text sep)</code>	converts into text with sep between elements.
text	<code>key()</code>	returns the key of the pointed out item.
void	<code>kSort()</code>	orders the indices, associations being preserved.
boolean	<code>load(text name)</code>	loads the file into the array.
dyn	<code>min()</code>	gets the lowest value.
dyn	<code>max()</code>	gets the highest value.
void	<code>pack()</code>	Makes elements of the array consecutives.
dyn	<code>pop()</code>	gets and removes the last item.
dyn	<code>pop(int)</code>	gets and removes the item at the given position.
void	<code>push(dyn)</code>	adds an item at end.
dyn	<code>rand()</code>	returns an item at random location.
list	<code>reverse()</code>	returns the list in reverse order.
dyn	<code>shift()</code>	gets and removes the first item.
int	<code>size()</code>	returns the number of elements.
void	<code>sort()</code>	sorts the values in ascending order.
void	<code>store(text name)</code>	stores the array into a file.
number	<code>sum()</code>	calculates the sum of items.
void	<code>unique()</code>	removes doubletons from values, first found kept.
void	<code>unShift(dyn)</code>	inserts an item at the first position.
dyn	<code>value()</code>	gets the value pointed out. <code>k,v = d.key(), d.value()</code> to read a pair key:value

Node: [Typed arrays](#), Next: [Constructor and literal](#), Previous: [Methods of array and dict](#), Up: [Top](#)

Typed arrays

Typed arrays hold a unique type of objects. They are very more efficient than common arrays because processing directly 32 bits or 64 bits objects or pointers rather than dynamic variables is faster. (This is for binary executables, there is no difference in a PHP target.) Typed arrays are dynamic also, size is not limited, and boundary overflow is controlled.

Virtual methods of typed arrays are identical to that of mixed arrays. See above.

Node: [Constructor and literal](#), Next: [Array of int](#), Previous: [Typed arrays](#), Up: [Top](#)

Constructor and literal

The constructor of a typed array has the form:

```
type(...elements...)
```

Example:

```
int(1, 2, 3)
```

The number of element is between 0 and n.

When there is only one element, there is no difference between the constructor of typed array and the constructor of a scalar:

```
int(1)
text("text")
```

This is not a problem at assignment as we can create an array from a scalar, but inside an expression this is ambiguous. Inside an expression we must use a literal array rather than a constructor:

```
type{...elements...}
```

Example:

```
int{ 1, 2, 3 }
```

The curly braces mean for "array of".

Node: Array of int, Next: [Array of text](#), Previous: [Constructor and literal](#), Up: [Top](#)

Array of int

The syntax to declare an array of int is:

```
int[] i = int(x, y, ....)
```

Once it is declared, it is used as a common mixed array, but only integer numbers may be added to the array.

At creation stage, a constructor or a literal may be assigned.

If you use a single argument: `int(10)`, for example, this can be assigned either to a variable or to an array of int. In an expression a such constructor is a scalar. You must use a literal for one-element typed array.

Examples:

```
int[] x = int(5)
int[] x = int(1,2)
int[] x = y + int{5}
```

Node: Array of text, Next: [Array of numbers](#), Previous: [Array of int](#), Up: [Top](#)

Array of text

The declaration is:

```
text[] t = text(a, b, "demo", ...)
```

The rules of int array apply to text array also and all type of typed array.

Node: Array of numbers, Next: [Array of objects](#), Previous: [Array of text](#), Up: [Top](#)

Array of real, natural, number

The declarations are:

```
real[] r = real(1.0, 2.0, ...)
natural[] n = natural(1, 2, ...)
number[] n = number(1, 2.0, ...)
```

Node: Array of objects, Next: [Assignment and conversion](#), Previous: [Array of numbers](#), Up: [Top](#)

Array of objects

The declaration is:

```
class Someclass
    ...
/class

Someclass object1                ... instance
Someclass object2                ... instance
object[] = object(object1, object2, ...) ... array of objects
```

Node: Assignment and conversion, Next: [Using typed array with dyn](#), Previous: [Array of objects](#), Up: [Top](#)

Assignment and conversion

You can assign a typed array to a mixed array:

```
int[] i = int(...)
array a = i
```

- You can't, of course, assign a mixed array to a typed array since the values may have different types, unless you push the elements one by one. A literal array without prefix is considered to be a mixed array. Thus you can't assign it to a typed array.

Examples:

```
{1,2,3}                ... this is a mixed array even with integer items.
int[] i = {1,2,3}      ... NOT VALID you must write instead:
int[] i = int{1,2,3}   ... valid.
```

- You can't create a mixed array with a constructor of typed array:

```
array a = int(1,2)     ... NOT VALID
array a = array(int(1,2)) ... NOT VALID
```

Node:Using typed array with dyn, Next:[Random assignment](#), Previous:[Assignment and conversion](#), Up:[Top](#)

Using typed array with dyn

You can assign a typed array to a dyn.

```
int[] i = int(1,2)
dyn d = dyn(i)
```

Two new dyn methods are provided to use a such dyn:

`arrayType()`

return the type of the typed array assigned to a dyn.

`toList(int = 0)` return a typed array assigned to a dyn.

The parameter is the type (see below) and is required only if the dyn hold a classical array we want return as a typed one.

Types currently recognized are:

```
$TYPE_INT          array of integers
$TYPE_TEXT         ...
$TYPE_REAL
$TYPE_NATURAL
$TYPE_NUMBER
$TYPE_OBJECT
```

Example of use:

```
d = dyn(int(1,2))
if d.arrayType()
= $TYPE_INT:
    int[] i = d.toList()
= $TYPE_TEXT:
    text[] t = d.toList()
/if
```

Node:Random assignment, Next:[Limitations and compatibility](#), Previous:[Using typed array with dyn](#), Up:[Top](#)

Random assignment

For compatibility with PHP, adding an element to an array outside the boundary, has same effect than pushing it on top of the array.

For example:

```
int[] i
i[10] = 10          ...this is equivalent to: i.push(10)
```

Pushing values is the right way to fill an array in fact.

If you want really putting element at fixed position, you have to fill the array with some values...

```
for int k in 0 .. 10 let i.push(0)
```

you can now put an element at the 10 indice.

Node:Limitations and compatibility, Next:[File](#), Previous:[Random assignment](#), Up:[Top](#)

Limitations and compatibility

- You can't apply a method directly to an element of a typed array.

```
i[10].toText()    .. not valid
int x = i[10]
x.toText()       .. valid
```

- You can't compare directly an element and a string.

```
text[] t = text("one", "two")
if t[0] = "one"           ... not valid
text t2 = t[0]
if t2 = "one"             ... valid
```

- With PHP 4.3, inserting an object inside an array causes the values of attributes inserted, not the object itself! I consider this is a bug and so you can't insert directly objects, but you can insert an array of objects, even with a single object inside, instead.

- The PHP's `array_diff` function doesn't work with arrays holding objects, so subtracting such arrays is not possible.

Node:File, Next:[Read](#), Previous:[Limitations and compatibility](#), Up:[Top](#)

File

File is virtual object, for processing local or distant files.

For a complete explanation of the file system, see "fopen" in a C manual. A file object is created by an instance declaration. The file itself is created by the "open" command. The open command allows to access a file in different modes, according to the second parameter of the "open" method. The error control structure is used to test if the file has been properly opened or not.

Syntax to create or open a file:

```
file fname           declares a file.
fname.open(path, mode) opens the file with the path, and the mode.
```

Path types:

```
http://path         http distant file.
ftp://path          ftp distant file.
path                local file.
```

Modes:

```
"r"                read only.
"w"                write only.
"a"                append at end of file, write only.
```

"r+"	reads or write at start of file.
"a+"	reads at current location, or write at end.

File's methods are:

return	method	action
int	eof()	return true, if end of file reached.
int	open(text, text)	create or open a file. Set the error flag.
int	close()	close the file.
text	read(int)	read a chunk of size in argument, returns a text.
text	readLine()	read the next line from a text file, returns it.
int	seek(int)	go to the position in argument. Return 1 if ok.
int	size()	return the size of the file.
int	time()	return the time of last change.
int	write(text)	write the text in argument, return the size
written.		
void	writeLine(text)	write the text in argument.

Examples:

file myfile	creates a file object.
myfile.open("myfilename", "r")	opens a real file.
myfile.close()	closes the file.
boolean b = myfile.eof()	returns true if end of the file reached.
myfile.seek(250)	reads or writes skipping the first 250 bytes.

Distant files are not handled by Scriptol C++ and the interpreter for now.

Node:Read, Next:[Write](#), Previous:[File](#), Up:[Top](#)

Read

A file may be read line per line, or by binary blocks.

In the first case, the readLine() method is used, otherwise the read method has the size of the block as parameter.

The end of line code is included in the text and may be removed by the rTrim() text method.

Examples:

text t = fname.readline()	reads a line terminated by a newline code.
text t = fname.read(1000)	reads a block of 1000 bytes from the file.

Node:Write, Next:[Dir](#), Previous:[Read](#), Up:[Top](#)

Write

The write method puts the content of a text variable inside a file. Once the file is open (either in "w" or "a" mode), and thus written either from scratch or at end of the current content, each new call to the write method results in appending a new text at end.

Example:

text filename = "myfile"	
file outfile	name of the virtual object.
outfile.open(filename, "w")	opens to write.

```
error? die("enable to open " + filename)
for text line in outList
  outfile.write(line)
/for
outfile.close()
```

```
if error, exits with a message;
an array previously filled
writes an element

closes the file
```

Node:Dir, Next:[The error control structure](#), Previous:[Write](#), Up:[Top](#)

Dir

The dir type has two methods, the constructor and get(), and allows to use built-in directory management functions.

The constructor creates a dir or return it to be displayed.

```
void dir(text)          constructor. The argument is a path
array get()            returns the content of the directory into a array.
```

```
print dir("c:/games")
```

or

```
dir x = dir("subdir")
print x
```

or

```
array a = x.get()
a.display()
```

The directory separator is "/" for any system.

Directory management functions are:

```
dir opendir(text)      opens a directory, returns a dir.
text readdir(dir)     reads next entry and returns the name.
void closedir(dir)    closes the directory.
void rewinddir(dir)   moves to the first entry.
boolean is_dir(text)  returns true if the entry is a directory.
boolean is_file(text) returns true if the entry is a file.
```

The functions chdir, mkdir, rmdir are detailed in the chapter on built-in function.

Example of use:

```
dir mydir = opendir("/path")
while forever
  text t = readdir(mydir)
  if t = nil break
  if t[0] <> "." print t
/while
```

Node:The error control structure, Next:[Scopes](#), Previous:[Dir](#), Up:[Top](#)

The error control structure

After the "open" statement, the control structure "error /error" or one-line "error ?" should be executed to detect when an access error occurs. But the interpreter may stop the program before the construct is processed.

Otherwise the body of the error block is processed when an error occurs.

The syntax is:

```
xxx.open("filename")
error
  ...statements...
/error
or:
xxx.open("filename"); error ? action
or:
xxx.open("filename"); error action
```

Example:

```
file myfile
myfile.open("filename", "r")
error? exit()
```

If the file can't be found or opened, the program exits.

Node:Scopes, Next:[Extern](#), Previous:[The error control structure](#), Up:[Top](#)

Scopes

Rules of visibility of variable are that of main procedural language, not that of scripting languages. Scriptol add some safety rules, it doesn't allow a same name to be given to different objects in embedded scopes (for example, the global scope and the one of a function). But names may be reused in successive scopes.

There are four scoping levels: - global, - class, - body of a function, - and body of a control structure.

The scope of a variable is level where it is declared: global, class, function or delimited block of statements (body of if, while, do, case, etc...), and inner levels.

A variable can't be redeclared inside the same scope, but only when the scope is closed, inside other scopes.

The header of the for control structure is a part of the body's scope.

```
for text t in a          t is visible only inside the for loop
  t = a.inc()
/for
```

The scope of the let statement that ends a while control structure is a part of the body of the while structure.

Arguments of a function are inside the scope of the function.

Inside a function, if a variable is referenced before any assignment, it is assumed referencing a global variable if it exists, otherwise this is an error.

Inside a method of a class, if it is referenced before any assignment, it references to a member if this

member exists, otherwise this is an error. Global variables are not visible inside classes, but the external ones.

External variables (those of PHP, Apache, etc...) are always in the scope, since there is no control for them (look at the "External variable" section).

Thus, you have to know their name to avoid using a name that will become that of a PHP variable, when the "\$" will be added to.

If you use PHP external variables inside a function, you must declare them as "global", as the compiler does not manage them.

Node:Extern, Next:[Variables and constants](#), Previous:[Scopes](#), Up:[Top](#)

Extern, external variables, constants and functions

Functions, variables and constants of the PHP or C++ languages or extensions may be used inside the Scriptol code.

Their scope is external, that means they are visible anywhere, in global, local or class scopes.

Node:Variables and constants, Next:[Functions](#), Previous:[Extern](#), Up:[Top](#)

Variables and constants

PHP variables and constants may be used as scriptol objects, providing they are declared with the extern keywords.

Extern declarations, as include ones, must be put at start of a source file.

The syntax is:

```
extern type identifier
extern constant type identifier
```

No any test is performed on the type and the scope or external, and you must also use a global statement if you use them inside a function.

Example:

```
global argv
array a = $argv
```

A PHP constant is declared as a variable, with the "constant" modifier.

Example:

```
extern constant text PHP_VERSION
...
print PHP_VERSION
```

It is possible also to use a PHP or C++ variable without declaration with the \$ prefix followed by an

identifier.

It can start with an underscore.

<code>\$var_name</code>	
<code>\$(constant_name)</code>	in PHP
<code>\$constant_name.</code>	in C++

Node:Functions, Next:[External classes](#), Previous:[Variables and constants](#), Up:[Top](#)

External functions

PHP functions may be used without declaration, this is not the case for C++ functions.

To declare a external function, a extern statement is required.

A default value is denoted by an assignment. The value is used in the C++ target, it is ignored by PHP.

Syntax and example:

```
extern type identifier(... arguments...)  
  
extern text substr(text, int, int = 0)
```

Node:External classes, Next:[External types](#), Previous:[Functions](#), Up:[Top](#)

External classes

To use methods of PHP or C++ classes, you must declare the classes and the members you want to use.

Example:

```
extern  
  class phpClass  
    int phpAttribute  
    void phpMethod(int argument)  
    ... declarations ...  
  /class  
/extern
```

Node:External types, Next:[Inserting native code](#), Previous:[External classes](#), Up:[Top](#)

External types

C++ allows to create new types with the typedef and define directives. These types may be integrated into a Scriptol source with the "define" instruction:

Example: define uint

This corresponds to "#define uint unsigned int" in C++.

No any code is generated by the define instruction: the C++ definition must existats inside an included C++ header file.

See at "The define statement" for more on this command.

Node:Inserting native code, Next:[Class](#), Previous:[External types](#), Up:[Top](#)

Inserting native code

If you want to insert PHP or C++ code directly into your program, use the `~~` symbols to enclose the code.

The compiler doesn't process it, it is as a comment for the Scriptol compiler, but it is processed by the target interpreter or compiler.

You can include code specifically for the PHP interpreter or C++ compiler.

- If you want to insert some PHP code, use this statement: `require_once("thefile.php")` This statement is interpreted in PHP, but is ignored by the C++ compiler.

- To insert C++ code, use a such statement: `#include "thefile.cpp"`

This is compiled in C++, but for the PHP interpreter this is just a comment.

Node:Class, Next:[Defining a class](#), Previous:[Inserting native code](#), Up:[Top](#)

Class

A class is a structure which contains variables (attributes) and has functions (methods).

Once a class is defined, it becomes a new type added to the language and unlimited number of instances (objects) may be declared with this new type. We use attributes and methods of the object with a command in the form:

```
x = object.attribute
object.method()
```

Node:Defining a class, Next:[Constructor and instance](#), Previous:[Class](#), Up:[Top](#)

Defining a class

The description of a class begins with the "class" keyword and ends with "/class".

Example of a class declaration:

```
class car
  ...members...
/class
```

Example of class with attributes and methods:

```
class car
  int theSpeed = 50
  int passengers = 4

  void car.speed(int i)
    int x = theSpeed * 2
    return x
/class
```

Example of instance and method reference:

```
car mycar
print mycar.passengers
print mycar.speed()
```

Node:Constructor and instance, Next:[Static methods and attributes](#), Previous:[Defining a class](#), Up:[Top](#)

Constructor and instance

A constructor is a method the name of which being that of the class, and that is called only when creating an instance of the class. The constructor always returns a void type.

Example of constructor:

```
class Car
    int speed

    void Car(int p)           ` this method is the constructor
        speed = 0           ` speed is initialized with a default value
    return
/class
```

The syntax to create an instance is:

Classname instancename = Classname(argument [, arguments])

or:

Classname instancename ... for a constructor without arguments

Example:

```
Car mycar = Car(60)           ... create a car with a speed of 60
Truck mytruck                 ... is equivalent to
Truck mytruck = Truck()
```

Node:Static methods and attributes, Next:[Inheritance](#), Previous:[Constructor and instance](#), Up:[Top](#)

Static methods and attributes

It is convenient to store all functions relative to a task into a class and declare them "static". You can then refer to these methods directly along with the class name without the need for declaring an instance.

Example:

```
node, ext = Dir.splitExt(path)
```

Static attributes are also allowed . A static attribut is common to all instances of the class and inherited ones. Static methods can use only static attributes, as other ones exist only in instances of the class, Static methods can't reference other methods.

The "static" modifier must precede the type of the object. An error message is thrown if a static method attempts to use a non-static attribute or to call another method.

Example of static attributes and methods:

```

class Car
    static factory = "xxx"
    static void setName(text x)
        factory = x
    return
/class

Car.setName("name")
Car.factory = "name"

```

Static attributes and methods may be associated to an instance also:

```

Car myCar
myCar.setName("name")
myCar.factory = "name"

```

The result will be the same.

Node: [Inheritance](#), Next: [Overloading](#), Previous: [Static methods and attributes](#), Up: [Top](#)

Inheritance

A class may inherit from attributes and method of another one, if it is declared as a subclass of it. The class "name" inherits of attributes and methods of the superclass "othername". This works also for static attributes and methods.

The syntax of inheritance declaring is:

```
class name is othername
```

Example:

```

class Vehicle
    int fuel
    void Vehicle()
        fuel = 100
    return
/class

class Car is Vehicle
    int passengers
    void Car()
        passengers = 3
        fuel = 50
    return
/class

class Truck is Vehicle
    int charge
    void Truck()
        fuel = 200
        charge = 1000
    return
/class

```

```

Truck redbaby
print redbaby.charge
print redbaby.fuel
print redbaby.passengers
Car daisy

```

```

attribute of the Truck class
attribute of the Vehicle superclass
bad! passengers is not accessible to Truck!

```

Node:Overloading, Next:[Static XML](#), Previous:[Inheritance](#), Up:[Top](#)

Overloading

A method may be redefined with different arguments, both inside the same class or into inherited classes. The return type must be the same for any definition of the method.

Example:

```
void add(int x, int y)
void add(real x, natural y)
```

You have just to call the "add" method with any arguments, the compiler associates the corresponding definition...

The main target language, C++ requires the return type remains the same, and this has been kept also in Scriptol.

Node:Static XML, Next:[Light XML](#), Previous:[Overloading](#), Up:[Top](#)

Static XML

NOTE: This chapter, Static XML, and the follower are supported by the interpreter only for now.

The Scriptol to binary compiler uses a dom class with similar methods and requires the libsol.sol file, that defines the class.

The dom class is described in dom.html.

The Scriptol to PHP compiler doesn't handle XML for now.

In the future, this chapter will apply to Scriptol compiler also.

XML is a data structure in Scriptol. An XML document is defined in light form, and then used to store and deliver data, just as classes, with more flexibility.

The XML document may be used statically, only the content of elements and values of attributes are changed.

An element is accessed by a chained list of name of elements that hold it.

```
dom.document.customer.getData()
```

Methods are provided also to change the structure, add or remove elements. This is covered in the "Dynamic XML" section.

Node:Light XML, Next:[Defining an XML document](#), Previous:[Static XML](#), Up:[Top](#)

Light XML

XML is written in scriptol source in a light syntax, without angle-brackets.

The syntax of an XML structure is:

```
xml [name]           ... optionally a name for the document.
  [headers]         ... optional header elements.
  [doctype]         ... optional doctype element.
  [include]         ... optionally one or several include statement.
  tag               ... one or several tags.
/xml
```

Include

The include directive currently allow to include another xul document.

Syntax:

```
include href = "other-xul-document-location"
```

Header

Syntax:

```
<? identifier and attributes ?>
```

Doctype

Syntax:

```
<! identifiers and attributes >
```

Note that the ending is > and not !>

Element

A tag has the form:

```
tagname [attribute [[,] attribute]* ]   optional a list of attributes.
  [tagname2]*                          none, one or several sub-elements.
  text                                  or a text
  ...
/ tagname                               the tag name is optional
```

If the list of attributes is continued on the next line, a comma must end the current line at least.

Node:Defining an XML document, Next:[Loading and saving](#), Previous:[Light XML](#), Up:[Top](#)

Defining an XML document

An XML document is declared as a class, enclosed between the xml and /xml markers, in light format.

```
xml docname
  tagname [attributes]
    "data"
  or
```

```
    ~~
      several lignes of data
    ~~
    or
      embedded elements
  / tagname
/xml
```

XML instance may be declared then.

```
docname instance1
docname instance2
```

Instances are created at compile time. Changes in the document at run time are not retrieved into instances.

Node:Loading and saving, Next:[XML assignment](#), Previous:[Defining an XML document](#), Up:[Top](#)

Loading and saving

External XML documents are loaded into a Scriptol program with the "load" method, and a filename or path as parameter.

```
xml x
/xml
x.load("demo")
```

or:

```
x xinstance
xinstance.load("demo")
```

The document, loaded and updated, or defined inside a Scriptol source is saved with the "save" method and any filename or path as parameter.

Node:XML assignment, Next:[XML iterator](#), Previous:[Loading and saving](#), Up:[Top](#)

XML assignment

You can assign an instance with a document. You need to declare a master empty XML document to make instances, before to assign the instances.

Example using the "docname" document defined above.

```
xml dom
/xml

dom instance1 = docname
```

The content of instance1 is now that of the "docname" document.
From the "dom" document, you can declare how many instances you want.

XML iterator

Iterator is a set of method that can be used with tag methods.

Tag methods are context methods that are associated with the name of the document or the name of an instance of the document, but are related to the currently pointed out tag.

These methods allow to read and change data, attributes, and values of attributes.

The iterator allows to scan the content of a document or a instance, and is just a set of methods:

```
begin()           move to the first element at the current level.
inc()             point out the next element.
found()           return true if an element is found by inc() or down().
down()            move to the first embedded element.
up()              move to the parent.
```

The document may be scanned backward with these methods:

```
end()             point out the last tag at the current level.
dec()             go to previous tag at same level.
```

Example of iterator use:

```
demo.reset()
demo.down()
while demo.found()
    print demo.getData()
let demo.inc()
```

Iterator acts on a level of the document. To enter a level, the at() method is used, along the chained list of level container's names.

```
xml demo
  document
    element
      "data'"
    /
    element
      "data"
    /
  /
/xml

demo.document.element.at()
```

We are at "element" level, held by the document level. The reset() method restart the current location at top of the document.

Using data

Data are used with two tag methods, getData and setData that allow to access the content of an element:

```
text getData()          return the content.
void setData(text)     change the content.
```

These methods are associated by a dot to the name of a document or an instance, and act on the currently pointed out element.

```
docname.getData()
instance1.setData("some text")
```

Node:Using attributes, Next:[XML in functions](#), Previous:[Using data](#), Up:[Top](#)

Using attributes

Attributes are accessed as dictionary items. They are recognized by their name and accessed with the `getAttribute()` and `setAttribute()` methods.

```
xml doc
  elem1 att = "value"
/elem1
/xml

print doc.getAttribute("att1")
doc.setAttribute("att1", "new value")
```

Node:XML in functions, Next:[Dynamic XML](#), Previous:[Using attributes](#), Up:[Top](#)

XML in functions

You can use instance of XML document as argument of a function, and return it from a function. The argument is a new instance, not a pointer on the original instance, and the object returned is also a new instance.

An XML document can be defined inside a function. If the function return it, this returns a pointer on the XML document, to improve speed.

```
xml dom
/xml

dom myfunction()
  xml demo
    tagname "content"
  /
/xml
return demo

dom x = myfunction()
print myfunction()
```

Node:Dynamic XML, Next:[XML methods](#), Previous:[XML in functions](#), Up:[Top](#)

Dynamic XML

This chapter describes function to change the structure of an XML document or an instance.

`void clear()`

Remove the content the document.

`xml addChild(xml)`

Add a child to the current element of the document.

The new element becomes the current element for the document.

```
xml demo
  doc /
/xml

xml element
  inner
  "some text"
/inner
/xml

demo.addChild(element)
```

`xml addNext(xml)`

Add an element to the currently pointed out element in the document.

The new element becomes the current element for the document.

Node:XML methods, Next:[Included files](#), Previous:[Dynamic XML](#), Up:[Top](#)

XML methods

Methods

Definition

<code>xml addChild(xml)</code>	Add a child to the current element in the document.
<code>xml addNext(xml)</code>	Add a successor to the current element in the document.
<code>xml append(xml)</code>	Append another document, return the current augmented document.
<code>xml at()</code>	Returns currently pointed out element, as an XML document.
<code>xml at(text)</code>	Point out an element in the current level, its name is the argument.
<code>xml at(text, text)</code>	Point out the tag with attribute name and values in arguments.
<code>xml begin()</code>	Move to start of level. Practically, go to first child of parent of current tag.
<code>void clear()</code>	Erase the content of the document.
<code>xml clone(xml)</code>	Copy the document into the document in argument.
<code>xml dec()</code>	Move to previous tag.
<code>void display()</code>	Display the document.
<code>xml down()</code>	Go to the first child of the current tag.
<code>boolean empty()</code>	Returns true if the document holds no element.
<code>xml end()</code>	Go to the last tag at the current level.
<code>boolean found()</code>	Return true or false if the inc or dec method are successful or not.
<code>text getData(text)</code>	Return content of a tag.
<code>text getValue(text)</code>	Return the value of the attribute in argument (for pointed out tag).

```

boolean hasAttribute(text) Return true
xml inc() Go to next tag.
int length() Return the number of tag at the current level.
boolean load(text) Fill the xml structure with an xml document from a
file. Return true if ok.
xml reset() Go to the first element of the document.
boolean save(text) Save the xml document into a file. Return true is
successful.
void setData(text) Change content of a tag.
void setValue(text, text) Change value of an attribute. Or create it.
xml up() Go to the successor of the parent of the currently
pointed out tag.
void xml([xml]) Constructor. May optionally take another document as
argument.
-----

```

Node:Included files, Next:[The define statement](#), Previous:[XML methods](#), Up:[Top](#)

Included files

The syntax to include an external scriptol file is:

```
include "filename.sol"
```

Parenthesis are optional. Simple or double quotes are required. If you want to use directly a PHP file, and the compiler not to compile it, see below...

Only file having a ".sol" extension are parsed by the scriptol compilers. Otherwise:

- the solp (php) compiler does remove the statement and ignore it.
- the solc (c++) compiler passes the statement to the C++ compiler.

If, conversely, you want to pass an include command just to the PHP code, use a `require_once("filename")` statement, it will be ignored by the solc (c++) compiler, as dynamic include is not allowed in C++.

Node:The define statement, Next:[Using a function as a variable](#), Previous:[Included files](#), Up:[Top](#)

The define statement

This statement allows to define new types the parser can recognize along with built-in primitives and classes.

Node:Using a function as a variable, Next:[Using an external type](#), Previous:[The define statement](#), Up:[Top](#)

Using a function as a variable

You can use a function as argument of another function by defining this function as a type. This works only at the global level and can't be used inside a class. Once a type is defined, you can't redefine it.

1) define a function:

Example:

```
int compare(int a, int b)
    boolean b = (a < b)
    return b
```

2) define a type, using this function as model:

```
define COMP = "compare"
```

3) define another generic function that uses this function as argument:

```
void myfunc(COMP a, int x, int y)
    print a(x,y)
    return
```

4) use the generic function:

The argument may be the original function or another function with same arguments and return type.

```
myfunc("compare2", 10, 8)
```

You can now define another function, "mul" for example, with the same model, thus having same return type and same parameters that the "add" function, and use it instead.

```
myfunc("mul", x, y)
```

Node:Using an external type, Next:[Using standard libraries](#), Previous:[Using a function as a variable](#), Up:[Top](#)

Using an external type

The syntax is :

```
define NEWTYPE
define NEWTYPE as
```

creates a new type you can use in arguments of external functions. Look at the GTK examples for how to use the new type.

The "as" modifier prevents the new type to be used as a pointer. Use as if the type is a primitive, as uint or gint for example.

Node:Using standard libraries, Next:[Using PHP libraries](#), Previous:[Using an external type](#), Up:[Top](#)

Using standard libraries

Node:Using PHP libraries, Next:[Using C libraries](#), Previous:[Using standard libraries](#), Up:[Top](#)

Using PHP libraries

Scriptol can use PHP's functions. No control is performed on arguments and return type of PHP

functions.

You have just to configure the the php.ini file to make the extensions visible for the PHP interpreter. The most frequently used PHP's functions are implemented into the libsol library.

To use PHP extensions in a binary executable, you need for an interface that translates PHP names to original ones, as phpgd.h does, and eventually a wrapper from php code to the library, as phpgd.cpp.

Node:Using C libraries, Next:[Useful functions](#), Previous:[Using PHP libraries](#), Up:[Top](#)

Using C libraries

A C or C++ library may be used directly from a scriptol program just by adding a ".lib" or ".a" or ".so" file to the list of libraries into the ".ini" or ".cfg" configuration file.

A file to include have to be written to let variables and classes of the extension visible for the compiler, and are declared as extern.

The "define" statement allows to make a type, and use a function as argument of another function. C variables may be generated directly also:

```
extern
    ~~extern char *message~~      for the header file.
    char *message                 for the visibility in scriptol.
/extern

~~char *message~~               creating actually the C variable.
```

Equivalences:

C	Scriptol
char *	cstring
void *	object
char **	cstring *
unsigned char	byte

A function argument funname is written "funname" in Scriptol (see define).

A built-in function can't be used as argument. Encapsulate it into a user-defined function instead.

If an instance is declared in the C++ form ClassName instance, you have to create a pointer instead (the C++ form being ClassName *instance).

Node:Useful functions, Next:[Integrated XML](#), Previous:[Using C libraries](#), Up:[Top](#)

Useful functions

These functions are common to PHP, C, C++, and Scriptol.

text chr(number)	Returns the character for an ASCII value, ex "A" for 65.
void die(text)	Displays a message and exits the program.
void exit()	Exits the program.
number min(num, num)	Returns the lowest of two arguments.
number max(num, num)	Returns the greatest of two arguments.
int ord(text)	Gets the ASCII value of a character.
constant cstring plural(int)	Returns "s" if the integer is greater than 1.

array range(int, int)	Generates an array of integers from x to y.
cstring str(number)	Converts a number into a string of chars.
void swap(dyn, dyn)	Exchanges the content of two variables.
text pad(text t, len l [, text c][, int o])	Pads a text with blank space or the given string of chars.
t: text to pad. l: length to reach. c: char to add, default blank space.	
o: options STR_PAD_LEFT, STR_PAD_BOTH, default at right.	

File functions (see also at File type methods):

void exec(text)	Passes a command to the operating system.
bool file_exists(text)	Test if the file whose name is in argument, exists.
number filesize(text)	Return the size.
number filetime(text)	Return the time (use date function to display).
text filetype(text)	Return "file" or "dir".
boolean rename(text, text)	Renames a file. Returns false if impossible.
void system(text)	Passes a command to the operating system.
boolean unlink(text)	Deletes a file. Returns true if deleted.

Directory functions:

boolean chdir(text)	Changes the current directory. Returns false if unsuccessful.
boolean mkdir(text)	Creates a sub-directory, returns true if created.
boolean rmdir(text)	Deletes a sub-directory. Returns true if deleted.
text getcwd()	Returns the path of the current directory.

Math functions:

number abs(number)	Returns the absolute value of a number.
real acos(real)	
real asin(real)	
real atan(real)	
number ceil(number)	Returns the rounded up integer.
real cos(real)	
real exp(real)	
number floor(number)	Returns the rounded down integer.
number fmod(number, number)	Return the modulus of two numbers.
real log(real)	
number pow(number, number)	Returns the n power of a number.
int rand()	Returns a random number.
void randomize()	Starts a sequence of random numbers.
real round(real)	Round at nearest of floor or ceil.
real sin(real)	
number sqrt(number)	Returns the square root of a number.
real tan(real)	

Time functions:

int time()	Time in milliseconds since January 1, 1970.
dict localtime()	Current time and date at call into a dictionary, see below.

Keys of the dict returned by localtime:

tm_sec	Seconds after the minute	[0,61]
tm_min	Minutes after the hour	[0,59]
tm_hour	Hours after midnight	[0,23]
tm_mday	Day of the month	[1,31]
tm_mon	Months since January	[0,11]
tm_year	Years since 1900	
tm_wday	Days since Sunday	[0,6]
tm_yday	Days since January 1	[0,365]

Node: Integrated XML, Next: [Appendix I](#), Previous: [Useful functions](#), Up: [Top](#)

Integrated XML (Scriptol C++ only)

This chapter will be replaced by the chapters "Static XML" and "Dynamic XML".

What is described here is still working in the Scriptol to C++ compiler and will remain compatible with the new simpler way to process XML implemented currently in the interpreter..

Dom

To use the XML document, an instance of dom must be created. The dom tree is then filled by the build() method:

```
dom mydom
mydom.build()
```

The complete list of dom's methods is given in the tutorial on CD, the main ones are described in this chapter.

To display the document, use the display() method.

To save it, use the save() methods, with the filename as argument.

In both cases, the dom.LIGHT argument allows to output it light.

You can assign a dom tree from another dom tree with the assign() method.

Path

Once the dom tree created, you can access the data and the attributes of tags with dom's methods.

But scriptol has a special syntax to point out an element inside the document. Tags and embedded tags are treated as objects and sub-objects:

The syntax is:

```
instance [.tagname]* [ attribute-name : value]? [.method-name(arguments)]?
```

- the name of the instance of dom,
- a dot followed by the name of the tag,
- this for each sub-tag,
- a couple attribute : value between square brackets if we want to select a tag among several ones with same name,
- a dot followed by a method and its arguments.

If we want just point out an element, the at() method is used. But we can add any dom's method to the path of an element. See at demoxml.sol for an example.

Reading and writing

You can read and change values of the attributes and the data.

- getValue(attribute-name) returns the value of the attribute as argument.
- setValue(attribute-name, value) assigns a value to an attribute.
- getData() returns the data.
- setData(text) change or assign the text given as parameter.

The data can be also assigned directly:

```
Ex: mydom.sometag.subtag.setData("some text")
```

or: `mydom.sometag.subtag = "some text"`

Iterator

The dom class can also use the document as a stack.
Several methods allows to browse a dom tree.

- `begin()` points out the first element of the document.
- `next()` moves to the next element at same level (inside the same element).
- `up()` moves to the successor of the containing element.
- `down()` points out the child of the current element, if one.
- `found()` return true if a next or child element exists, false otherwise.

Once an element pointed out, the previous methods are used to access the data.

Inserting and removing elements

You can insert elements either as child or as successor to a pointed out element, or remove a pointed out element.

Syntax:

```
instancename.path.addNext(xnode)
instancename.path.addChild(xnode)
instancename.path.remove()
```

To insert an element, you have to declare it into another XML document, and get it with the `getNode()` method.

You can move an element by inserting at a new location and removing it then.

Html

A html page has this format:

```
xml
  html
  ...content...
/html
/xml
```

Node:Appendix I, Next:[Appendix II](#), Previous:[Integrated XML](#), Up:[Top](#)

Appendix I: Using the Java API with Scriptol

Scriptol implements declaration of Java objects thanks to the "java" modifier, and providing the Java extension is activated in `php.ini` (see at the install card).

In the same way you include files the content of which you want to use, each Java class you use must be imported, with the full path according to Java's style.

Once a Java class is imported, you declare instances and call methods exactly as you do for Scriptol classes.

The Java source have to been compiled to be imported.

The syntax is:

```
import java classpath-classname
or
import
    ... class declarations ...
/import
```

If the word "java" begins the path, the compiler recognizes it and the modifier may be omitted:
Examples:

```
import java java.awt.Dialog      ...is a valid declaration
import java.awt.Dialog          ...is valid too
import java MyClass             ...is a valid declaration
import MyClass                  ...is not valid for a Java class
```

The instances are declared just as instance of Scriptol classes.

```
java.awt.Dialog myDialog      ... or simply
Dialog myDialog
MyClass myInstance
```

If "Dialog" is not used elsewhere in the program, you can omit the path.
Once the instance declared, it may be used as any scriptol object.

```
myDialog.setVisible(true)
myInstance.disp()
print myInstance.x
```

If you declare your own Java classes, they must stay in separated files, and the files must have the name of the class with the .java extension. It has to be compiled by javac.

Node:Appendix II, Next:[Appendix III](#), Previous:[Appendix I](#), Up:[Top](#)

Appendix II: Exceptions handling

Exception processing works only with the C++ and PHP 5 target languages.

Syntax:

```
extern
    class exception
        string what()
    /class
/extern

try
    ... some statements ...
catch(exception e)
    print e.what()
/try
```

Node:Appendix III, Next:[Appendix IV](#), Previous:[Appendix II](#), Up:[Top](#)

Appendix III: Foreign programming

The preprocessor allows to write scriptol program with keywords in any human language.

To use it, you need for:

- 1) a list of keywords.
- 2) the -t option at command line (this doesn't work with the -w option).
- 3) an entry inside the solc.ini or solp.ini file (or .cfg files) in the form:

Keywords=extension

The extension is a two letters code according to the language you use, example:

Keywords=fr ...for the french language.

The original list of keywords is in the keywords.en file. If no list is available in the language you want to use, you need to create it, with each foreign keyword followed by the english counterpart (see keywords.fr for an example).

If the ".ini" ou the ".cfg" file relying to a source has no Keyword entry, the -t option is ignored for this source and english keywords are expected.

Node:Appendix IV, Next:[Index](#), Previous:[Appendix III](#), Up:[Top](#)

Appendix IV: Deprecated syntax

This syntax is deprecated:

<- and -> symbols.

string keyword. Replaced by cstring.

char * external declaration. Replaced by cstring.

match operator and its definition.

Methods on numbers, they are useless.

Enclosing array definition or dict definition between (). Use {} or array() or dict().

The \$ prefix to PHP strings is deprecated.

Node:Index, Previous:[Appendix IV](#), Up:[Top](#)

Index

- about: [About this manual](#)
- alias: [Alias](#)
- and: [Expression](#), [Operators](#)
- application: [Scriptol project](#)
- argument: [Alias](#)
- arithmetical: [Operators](#), [Expression](#)
- arobase: [Quiet mode](#)
- array: [Expression](#), [Array](#), [Scanning a two-dimensional array](#), [Content of an array](#), [Making an array](#), [Dynamic variables](#), [Methods of array and dict](#), [Variables or primitives](#), [Typed arrays](#)
- assignment: [Assignment and conversion](#), [Assignment](#), [Random assignment](#), [Conditional assignment](#), [Simple assignment](#)
- associative: [Dictionary](#), [Sequence and list](#)

- at: [Quiet mode](#)
- attribute: [Static methods and attributes](#), [Defining a class](#), [Class](#)
- augmented: [Compound assignment](#)
- binary: [Compiling a Scriptol program to native](#), [Expression](#), [Operators on array](#), [Operators](#)
- block: [Scopes](#)
- boolean: [Variables or primitives](#)
- break: [Do case](#), [Break and continue](#)
- built-in: [Useful functions](#)
- c: [Using C libraries](#)
- c++: [Using standard libraries](#), [Compiling a Scriptol program to native](#), [Inserting native code](#)
- case: [Do case](#)
- class: [Scopes](#), [Class](#)
- classe: [External classes](#)
- command: [Interpreting a Scriptol program](#)
- commands: [Compiling a Scriptol program to PHP](#)
- comment: [Comment](#)
- compatibility: [Content of an array](#), [Appendix IV](#), [Indexing an array](#)
- compiling: [Compiling a Scriptol program to native](#), [Compiling a Scriptol program to PHP](#)
- composite: [Composite if](#)
- compound: [Compound assignment](#)
- conditional: [Conditional assignment](#)
- constant: [Enum](#), [Variables and constants](#), [Constant](#)
- constructor: [Constructor and instance](#), [Constructor and literal](#)
- content: [Using data](#)
- continue: [While let](#), [Break and continue](#)
- control: [Control structures](#)
- conversion: [Assignment and conversion](#)
- data: [Using data](#)
- dec: [Iterator](#)
- declaration: [Declaration](#)
- define: [The define statement](#)
- definition: [Defining a class](#)
- deprecated: [Appendix IV](#)
- description: [Overview](#)
- dict: [Variables or primitives](#), [Methods of array and dict](#), [Dictionary](#)
- dimension: [Multi-dimensional array](#)
- dir: [Dir](#)
- directory: [Dir](#)
- do: [Do extended syntax](#), [Do case](#)
- dollar: [Variable in string](#)
- dom: [Integrated XML](#)
- dyn: [Using typed array with dyn](#), [Dynamic variables](#)
- dynamic: [Using typed array with dyn](#)
- echo: [Print and echo](#)
- editor: [My first program](#)
- enum: [Enum](#)
- error: [The error control structure](#)
- escape: [Quotes and escaping](#)
- exception: [Appendix II](#)
- expression: [Expression](#)

- extern: [Extern](#)
- external: [Using standard libraries](#)
- features: [Features of the language](#)
- file: [Variables or primitives](#), [File](#)
- first: [My first program](#)
- flow: [Control structures](#)
- for: [For in](#), [Operators on array](#)
- foreign: [Appendix III](#)
- french: [Appendix III](#)
- function: [Using a function as a variable](#), [Function](#), [Using an external type](#), [Useful functions](#)
- Functions: [Functions](#)
- global: [Scopes](#)
- how to: [My first program](#)
- html: [Scriptol in html page](#)
- identifier: [Identifiers and keywords](#)
- if: [If](#)
- in: [Operators on array](#)
- inc: [XML iterator](#), [Iterator](#)
- include: [Included files](#)
- index: [Indexing an array](#)
- indexing: [Indexing a dict](#), [Indexing](#)
- indice: [Indexing an array](#), [Indexing](#)
- inheritance: [Inheritance](#)
- inner: [Scopes](#)
- input: [Input](#)
- instance: [Constructor and instance](#), [Class](#), [Array of objects](#)
- integer: [Variables or primitives](#), [Array of int](#)
- interpreting: [Interpreting a Scriptol program](#)
- intersection: [Operators](#), [Operators on array](#), [Expression](#)
- interval: [Interval and dictionary](#), [Interval in array](#)
- iterate: [Control structures](#)
- iterator: [XML iterator](#), [Iterator](#)
- java: [Appendix I](#)
- keyboard: [Input](#)
- keyword: [Identifiers and keywords](#)
- language: [Features of the language](#)
- let: [While let](#)
- library: [Using standard libraries](#), [Using PHP libraries](#), [Using C libraries](#)
- light: [Integrated XML](#)
- limitation: [Limitations and compatibility](#)
- list: [Sequence and list](#), [Expression](#)
- literal: [Literals](#), [Making a dict](#), [Constructor and literal](#)
- load: [Loading and saving](#)
- local: [Scopes](#)
- logical: [Operators](#), [Expression](#)
- loop: [Control structures](#)
- main: [Main](#), [Default values](#)
- member: [Defining a class](#)
- method: [Static methods and attributes](#), [Defining a class](#), [Class](#), [Methods on text](#)
- multi-dimensional: [Multi-dimensional array](#)

- multiple: [Multiple assignments](#)
- native: [Compiling a Scriptol program to native](#), [Inserting native code](#)
- natural: [Array of numbers](#)
- nested: [Scopes](#)
- nil: [Nil and null](#)
- not: [Operators](#), [Expression](#)
- null: [Nil and null](#)
- number: [Array of numbers](#), [Typed arrays](#), [Variables or primitives](#)
- object: [Class](#), [Defining a class](#), [Array of objects](#)
- operator: [Operators](#), [Operators on array](#), [Symbols](#)
- or: [Expression](#), [Operators](#)
- overloading: [Overloading](#)
- php: [Using standard libraries](#), [Inserting native code](#), [Content of an array](#), [Using PHP libraries](#), [Indexing an array](#), [Scriptol in html page](#), [Compiling a Scriptol program to PHP](#)
- polymorphism: [Overloading](#)
- precedence: [Precedence](#), [Operators](#), [Expression](#)
- primitive: [Variables or primitives](#)
- print: [Print and echo](#)
- project: [Scriptol project](#)
- quiet: [Quiet mode](#)
- quote: [Quotes and escaping](#)
- random: [Random assignment](#)
- range: [Range](#), [Subscripting](#)
- read: [Read](#)
- readline: [Read](#)
- real: [Variables or primitives](#), [Array of numbers](#)
- relational: [Expression](#)
- save: [Loading and saving](#)
- scalar: [Variables or primitives](#)
- scan: [Scan by](#), [Operators on array](#)
- scope: [Scope and function](#), [Scopes](#)
- scriptol: [Overview](#)
- source: [Interpreting a Scriptol program](#), [Scriptol source file](#)
- stack: [Using array as a stack](#)
- standard: [Using C libraries](#)
- start: [My first program](#)
- statement: [Statement](#)
- static: [Static methods and attributes](#)
- structure: [Control structures](#), [Scopes](#)
- subscripting: [Subscripting](#), [Indexing](#), [Range](#)
- symbol: [Symbols](#)
- tag: [XML iterator](#), [Using data](#)
- text: [Array of text](#), [Text](#), [Expression](#), [Variables or primitives](#), [Typed arrays](#)
- two-dimensional: [Scanning a two-dimensional array](#)
- type: [External types](#), [Variables or primitives](#), [The define statement](#), [Typed arrays](#), [Class](#)
- typed: [Typed arrays](#)
- union: [Operators on array](#), [Operators](#), [Expression](#)
- until: [Do until](#)
- using: [Using standard libraries](#)
- variable: [Using an external type](#), [Using a function as a variable](#), [Variable in string](#), [Variables and](#)

[constants](#), [Scopes](#), [Variables or primitives](#)

- visibility: [Scopes](#), [Scope and function](#)
- web: [Scriptol in html page](#)
- while: [While](#), [Do extended syntax](#)
- wrapper: [The define statement](#)
- write: [Write](#)
- XML: [XML methods](#), [XML iterator](#), [XML assignment](#), [Using data](#), [XML in functions](#), [Dynamic XML](#), [Using attributes](#), [Loading and saving](#), [Defining an XML document](#), [Light XML](#), [Static XML](#), [Integrated XML](#)