CODEWORKER
Parsing tool and Code generator
—
User's guide & Reference manual
Release
4.5.1

Cédric Lemaire

Email: `codeworker@free.fr`

CONTENTS

7 Quantifying scripts

Overview

CodeWorker is a scripting language distributed under the *GNU Lesser General Public License* and devoted to manipulate many aspects of *generative programming* as easy and intuitive as possible. *Generative programming* is a software engineering approach for producing reusable, tailor-made, evolvable and reliable IT systems with a high level of automation.

The scripting language adapts its syntax to the subject it has to handle: - an extended-BNF syntax (declarative part of the language) for recognizing the format of the specifications to parse, - a procedural language for manipulating easily parse trees (the only structured type admitted by 'CodeWorker'), strings, files and directories, - a JSP-like syntax (imperative part of the language), which facilitates the writing of template-based code generation.

Thanks to this syntax adaptation, the scripting language is able to easily: - acquire any kind of specification of the IT system to produce (often XML but not necessary), - generate source code in a classical way (as Rational ROSE), managing protected areas of text that accept hand-typed code, - expand a source file like the class-wizard of Visual C++ (generated text is inserted at specified markups), - translate from a format to another (LaTeX to HTML, XSL to CodeWorker, ... no limit), - transform a source file (to instrument a source file with profiling features, ...).

These tasks are executed in a straightforward process, with no binding to an external programming language and with no translation of requirements specification.

1.1   Building a parse tree

CODEWORKER provides two methods for performing a parsing:

- the **reading of tokens** is procedural,

- the **BNF description** is declarative, and conforms to a kind of BNF (the Backus-Naur Formalism represents a grammar in a particular syntax) extended with regular expressions,

During the parsing of files, CODEWORKER feeds an appropriate data structure that is called a *tree*, a *parse tree*. A tree is a convenient structure to represent a hierarchical set of nodes, as in XML for instance. The parse tree is shared both by the parse task, which takes in charge of populating the tree, and by the source code generation that will walk through it for generating text.

We suggest to use the file extension **".cwp"** for *extended-BNF* parse scripts.

1.2   A universal source code/text generation

Given a specification provided in any kind of format, CODEWORKER will generate source code or text as required in *template-based* scripts.

The source code generation can use three modes: generation, expansion or translation.

- **generation mode** is used to let the script produce the most part of the output file, processing a kind of *template-based* generation as it exists for a JSP or PHP script. Only some areas called *protected areas* in the vocabulary of CODEWORKER are preserved in the file. This philosophy has been adopted by some modeling tools that generate a skinny skeleton copiously interspersed with areas intended to the developer.

- **expansion mode** is used when the file is mainly written by hand, but small portions need to be generated. The points where to insert the code are called *markups* in the vocabulary of the scripting language. The *Class Wizard* of Visual C++ changes source code following this principle.

- **translation mode** is used when both parsing and source code generation are required to process a file. It arrives for processing:

    - a *source-to-source* translation: a file must be rewritten in a different syntax. For example, a LaTeX file might have to be translated in HTML.

    - a *program transformation*: a source file has to change for optimizing, refactoring, instrumenting or rewriting some portions.
    For example, a script could add a trace at the beginning of each function body of a JAVA or C++ source code. To do that, parsing will serve to discover function bodies, and source code generation will insert the C++ or JAVA code that implements the trace.

We suggest to use the file extension `".cwt"` for *template-based* scripts.

1.3   About the manual

Efforts are focused on improving the reliability of this documentation on examples and on the reference manual (except on English text, I'm afraid!).

A formal representation describes all functions and procedures that CODEWORKER provides, with their prototype and a short explanation and an example and the list of all-similar functions and procedures. This formal representation is used to generate source codes of CODEWORKER that handle parsing and C++ mapping and execution of each function and procedure of the scripting language. This formal representation that conforms to what CODEWORKER expects in terms of function/procedure prototypes, is reused to generate the LaTeX part of the reference manual that describes each of them. Examples are executed while generating the documentation to be sure they are correct, and to report an up to date output.

The chapter *getting started* is partially generated too, and the guarantee is given that every script runs successfully and that every example file has the last annotations. To warrant that, scripts are executed while generating the documentation, and example/script files contain some formatted comments just before lines to annotate. While including them into the chapter, their content is numerated line by line, and notes are extracted. Notes are written just after the content, and refer to the line they explain.

The documentation is written in LaTeX. The great advantage of LaTeX is that it offers a powerful text processing and that it is easy to manipulate for source code generation (text format instead of binary, and it accepts comments). Markups are inserted into the documentation at the points where generated text must be included. A markup is a special comment that CODEWORKER recognizes. This mode of code generation is an illustration of what is called *expansion mode* before.

Getting started

This chapter is intended to help you to discover the scripting language and how it may serve your software development process.

CODEWORKER is delivered with:

- an executable called `CodeWorker`, which runs into a shell and that requires options on a command line,

- a library called `CodeWorker.lib`, which may be linked to C++ applications for extending them with parsing and source code generation feature,

- some C++ headers that allow exploiting the library and that are available into the `"include"` directory,

Binaries are available into the `"bin"` directory.

The scripting language adapts its syntax to the nature of the tasks to handle:

- Acquiring the specifications of what to generate requires to be able to recognize format, either invented for answering as fine as possible to the particularities of the any kind of project or existing on the market and available on the *script repository* (see http://www.codeworker.org/ScriptsRepository.html, in constant improvement). A declarative language processes the scan of the format, following an extended BNF syntax ; it accepts the intrusion of procedural instruction to populate the parse tree.

- Manipulating internal data easily and executing instructions with an expressiveness similar to a classical system programming language. The procedural part of the language enables to take them in charge.

- Generating, expanding or tranforming text. The imperative part of the language offers a *template-based* syntax that accepts instructions to navigate into the parse tree and to take advantage of facilities brought by a programming language.

Example:

CODEWORKER allows saving time to implement source code, if it disposes of a detailed design. Let start with a tiny modeling language that only understands object types and that we create just for this example:

```
    // file "GettingStarted/Tiny.tml":
1 class A {
2 }
3
```

```
 4 class B : A {
 5 }
 6
 7 class C {
 8     B[] b
 9 }
10
11 class D {
12     A a
13     C[] c
14 }
```

LINE 1: we declare the class A, without attributes,
LINE 4: we declare the class B, which inherits from A,
LINE 7: we declare the class C that encapsulates an array of B instances,
LINE 11: we declare the class D that encapsulates an association to an instance of class A and an array of C instances,

2.1   The parse tree

The role of the parsing is to populate the parse tree. Let suppose that, for each class, we need of the following attributes:

- **name**: the name of the class, C for example,

- **parent**: the name of the parent if exists, A for class B for instance,

- **listOfAttributes**: an array that contains the description of encapsulated attributes, a and c into class D for instance,

The description of an encapsulated attribute will require:

- **name**: the name of the attribute, a into class D for instance,

- **class**: the name of the class it belongs to, A for attribute a for instance,

- **isArray**: true if the attribute is an array like c for example,

To discover the parse tree, we'll first populate it by hand. To do that, let run CODEWORKER in console mode:
```
CodeWorker -console
```

Type the following line into the console, and be careful not to forget the final semi colon:
```
insert listOfClasses["A"].name = "A";
traceObject(project);
```

The insert keyword is used to create new branches into the parse tree. The root is named **project**, but hasn't to be specified, and a sub-node (or *attribute*) *listOfClasses* has been added. This sub-node is quite special: it has to contain an array of nodes that describe classes. Items are indexed by a string and are stored into their entrance order; so, the node that takes in charge of describing the class A is accessed via *listOfClasses["A"]*. The string "A" is assigned to the attribute *listOfClasses["A"].name*.

The procedure `traceObject(project)` shows us the first-level content of the root: the attribute *listOfClasses* and all its entries (only `"A"` for the moment). Let populate the tree with the description of the class `B`:

```
set listOfClasses["B"].name = "B";
```

The `set` keyword is used to assign a value to an existing branch of the parse tree. If this branch doesn't exist yet, a warning notices you that perhaps you have done a spelling mistake, to avoid inserting new bad nodes. But the node is inserted despite of the warning. As the language isn't typed, it allows avoiding some troubles. Let's continue:

```
ref listOfClasses["B"].parent = listOfClasses["A"];
traceLine(listOfClasses["B"].parent.name);
```

The node *listOfClasses["B"].parent* refers to the node *listOfClasses["A"]*, so *listOfClasses["B"].parent.name* is similar to *listOfClasses["A"].name*. Let start filling in the tree for class `C`:

```
insert listOfClasses["C"].name = "C";
pushItem listOfClasses["C"].listOfAttributes;
local myAttribute;
ref myAttribute = listOfClasses["C"].listOfAttributes#back;
```

The `pushItem` assignment command is another way to add a new node into an array, where the item is indexed by the position of the node, starting at 0. The `local` keyword allows declaring a variable on the stack. This variable is also a parse tree, but not attached to the main parse tree `project`. For more commodities, this variable will refer to the last element of the attribute's list: *myAttribute* is shorter to type than *listOfClasses["C"].listOfAttributes#back*. Notice that the last element of an array is accessed via '`#back`'. Let complete the attribute b of class `C`: `insert myAttribute.name = "b";`

```
ref myAttribute.class = listOfClasses["B"];
insert myAttribute.isArray = true;
```

The keyword `true` is a predefined constant string that is worth `"true"`. The keyword `false` also exists and is worth an empty string.

Exercise:

Populate the parse tree with the description of class `D`.

2.2   Scanning our design with a BNF-driven script

Now, we'll describe the format of our tiny modeling language thanks to a BNF grammar (see paragraph 4.3.216 for more elements about it) like it is recognized by CODEWORKER :

```
    // file "GettingStarted/Tiny-BNF.cwp":
1  TinyBNF ::=
2      #ignore(JAVA)
3      [classDeclaration]*
4      #empty
5      => { traceLine("this file is valid"); };
6  classDeclaration ::=
7      IDENT:"class"
8      IDENT
9      [':'  IDENT ]?
10     classBody;
11 classBody ::= '{' [attributeDeclaration]* '}';
12 attributeDeclaration ::= IDENT ['[' ']']?  IDENT;
```

```
13 IDENT ::= #!ignore ['a'..'z'|'A'..'Z']+;
```

LINE 1: the clause `TinyBNF` takes in charge of reading our design,
LINE 2: blanks and comments are allowed between tokens, conforming to the JAVA syntax ('**/***' '***/**' and '**//**'),
LINE 3: the clause `classDeclaration` is repeated as long as class declarations are encountered into the design,
LINE 4: if no class anymore, the end of file may have been reached,
LINE 5: the '**=>**' operator allows executing instructions of the scripting language into the BNF-driven script; this one will be interpreted once the file will be matched successfully,
LINE 6: the clause `classDeclaration` takes in charge of reading a class,
LINE 7: the clause `IDENT` reads identifiers and the matched sequence must be worth `"class"`,
LINE 8: the name of the class is expected here
LINE 9: the declaration of the parent is facultative and is announced by a colon,
LINE 11: the clause `classBody` reads attributes as long as a it matches,
LINE 12: the clause `attributeDeclaration` expects a class identifier and, eventually, the symbol of an array, and the name of the attribute,
LINE 13: the clause `IDENT` reads an identifier, composed of a letter or more, which cannot be separated by blanks or comments (required by the directive `#!ignore`),

This *BNF-driven* script only scans the design ; it doesn't parse the data. Type the following line into the console to scan the design *"Tiny.tml"*:

```
parseAsBNF("Scripts/Tutorial/GettingStarted/Tiny-BNF.cwp", project,
"Scripts/Tutorial/GettingStarted/Tiny.tml");
```

Output:

```
this file is valid
```

But this script isn't sufficient enough to complete the parse tree.

2.3   Parsing our design with a BNF-driven script

We have to improve the precedent script, called now *"Tiny-BNFparsing.cwp"*, for building the parse tree that represents the pertinent data of the design:

```
    // file "GettingStarted/Tiny-BNFparsing.cwp":
1 TinyBNF ::= #ignore(JAVA) [classDeclaration]* #empty
2    => { traceLine("this file has been parsed successfully"); };
3 classDeclaration ::=
4    IDENT:"class"
5    IDENT:sName
6    => insert project.listOfClasses[sName].name = sName;
7    [
8       ':'
9       IDENT:sParent
10      => {
11         if !findElement(sParent, project.listOfClasses)
12            error("class '" + sParent + "' should have been
declared before");
```

```
13          ref project.listOfClasses[sName].parent =
project.listOfClasses[sParent];
14          }
15      ] ?
16      classBody(project.listOfClasses[sName]);
17  classBody(myClass :  node) ::=
18      '{' [attributeDeclaration(myClass)]* '}';
19  attributeDeclaration(myClass :  node) ::=
20      IDENT
21      ['[' ']']?
22      IDENT;
23  IDENT ::= #!ignore ['a'..'z'|'A'..'Z']+;
```

LINE 5: the name of the class is put into the local variable `sName`. Note that the first time a variable is encountered after a token, it is declared as local automatically.

LINE 6: we populate the parse tree as we have proceeded manually,

LINE 9: the name of the parent class is put into the local variable `sParent`,

LINE 11: the parent class must have been declared before: the item is searched into the list of classes,

LINE 13: we populate the parse tree as we have proceeded manually,

LINE 16: clauses may accept parameters; here, the current class is passed to `classBody` that will populate it with attributes,

LINE 17: the clause `classBody` expects a parameter as a node; a parameter may be passed as `value` or `node` or `reference`,

LINE 19: little exercise: complete the clause `attributeDeclaration` that takes in charge of parsing an attribute of the class given to the argument *myClass*,

LINE 20: remember that you must parse the class name of the association here (attribute *myClass.listOfAttributes#back.class* refers to the associated class),

LINE 21: remember that you must parse the multiplicity of the association here (attribute *myClass.listOfAttributes#back.isArray* is worth `true` if **'[]'** is present),

LINE 22: remember that you must parse the name of the association here (to put into attribute *myClass.listOfAttributes#back.name*),

Exercise:

Complete the precedent clause `attributeDeclaration` to populate an attribute. You'll find the solution into file *"Scripts/Tutorial/GettingStarted/Tiny-BNFparsing1.cwp"*.

Solution:

```
    // file "GettingStarted/Tiny-BNFparsing1.cwp":
1  classBody(myClass :  node) ::=
2      '{' [attributeDeclaration(myClass)]* '}';
3  attributeDeclaration(myClass :  node) ::=
4      IDENT:sClass
5      => local myAttribute;
6      => {
7          pushItem myClass.listOfAttributes;
8          ref myAttribute = myClass.listOfAttributes#back;
9          if !findElement(sClass, project.listOfClasses)
10             error("class '" + sClass + "' should have been
declared before");
11         ref myAttribute.class = project.listOfClasses[sClass];
12      }
```

```
13       ['[' ']' => insert myAttribute.isArray = true;]?
14       IDENT:sName => {insert myAttribute.name = sName;};
15
16  IDENT ::= #!ignore ['a'..'z'|'A'..'Z']+;
```

LINE 4: the name of the class for the association is assigned to the local variable `sName`,

LINE 5: we'll need a local variable to point to the attribute's node for commodity,

LINE 7: the local variable `myAttribute` hasn't been declared here, because it disappears at the end of the scope (the trailing brace); a new node is added to the list of attributes,

LINE 8: the local variable `myAttribute` points to the last item of the list,

LINE 9: the class specifier of the association must have been declared,

LINE 11: we populate the parse tree as done by hand,

LINE 13: this attribute *isArray* is added only if the type of the association is an array,

LINE 14: we complete the attribute description by assigning its name,

Type the following line into the console to parse the design *"Tiny.tml"*:

```
parseAsBNF("Scripts/Tutorial/GettingStarted/Tiny-BNFparsing1.cwp", project,
"Scripts/Tutorial/GettingStarted/Tiny.tml");
```

Output:

```
this file has been parsed successfully
```

2.4   Implementing a leader script

Now, we'll implement a little function that displays the content of our parse tree. We stop using the console here, and we'll implement the call to the parsing and the function into a *leader script*. This script will be called at the command line, as seen further.

We suggest to use the file extension **".cws"** for non-template and non-BNF scripts.

CODEWORKER *command line to execute:*
```
-script Scripts/Tutorial/GettingStarted/Tiny-leaderScript0.cws
```

```
      // file "GettingStarted/Tiny-leaderScript0.cws":
 1  parseAsBNF("Tiny-BNFparsing1.cwp", project,
"Scripts/Tutorial/GettingStarted/Tiny.tml");
 2
 3
 4  function displayParsingTree() {
 5     foreach i in project.listOfClasses {
 6        traceLine("class '" + i.name + "'");
 7        if existVariable(i.parent)
 8           traceLine("\tparent = '" + i.parent.name + "'");
 9        foreach j in i.listOfAttributes {
10           traceLine("\tattribute '" + j.name + "'");
11           traceLine("\t\tclass = '" + j.class.name + "'");
12           if existVariable(j.isArray)
13              traceLine("\t\tarray = '" + j.isArray + "'");
14        }
15     }
```

```
16  }
17
18  displayParsingTree();
```

LINE 4: a user-defined function without parameters,
LINE 5: the `foreach` statement iterates all items of an array; here, all classes are explored,
LINE 7: check whether the attribute *parent* exists or not,
LINE 9: all attributes of the current class *i* are iterated,
LINE 12: perhaps the association is multiple,
LINE 18: a call to the user-defined function,

Output:

```
this file has been parsed successfully
class 'A'
class 'B'
    parent = 'A'
class 'C'
    attribute 'b'
        class = 'B'
        array = 'true'
class 'D'
    attribute 'a'
        class = 'A'
    attribute 'c'
        class = 'C'
        array = 'true'
```

2.5   Generating code with a pattern script

The source code generation exploits the parse tree to generate any kind of output files: HTML, SQL, C++, ...

A *pattern* script is written in the scripting language of CODEWORKER extended to be able to fuse the text to put into the output file and the instructions to interpret. It enables to process a template-based generation. Such a script looks like a JSP template: the script is embedded between tags **'<%'** and **'%>'** or **'@'**.

We'll start by generating a short JAVA class for each class of the design. It translates the attributes in JAVA and it generates their accessors:

```
      // file "Scripts/Tutorial/GettingStarted/Tiny-JAVA.cwt":
1  package tiny;
2
3  public class @this.name@ @
4  if existVariable(this.parent) {
5      @ extends @this.parent.name@ @
6  }
7  @{
8      // attributes:
9  @
10  function getJAVAType(myAttribute :  node) {
```

```
11      local sType = myAttribute.class.name;
12      if myAttribute.isArray {
13          set sType = "java.util.ArrayList/*<" + sType + ">*/";
14      }
15      return sType;
16  }
17
18  foreach i in this.listOfAttributes {
19      @ private @getJAVAType(i)@ _@i.name@ = null;
20  @
21  }
22  @
23      //constructor:
24      public @this.name@() {
25      }
26
27      // accessors:
28  @
29  foreach i in this.listOfAttributes {
30      @ public @getJAVAType(i)@ get@toUpperString(i.name)@() {
return _@i.name@; }
31      public void set@toUpperString(i.name)@(@getJAVAType(i)@
@i.name@) { _@i.name@ = @i.name@; }
32  @
33  }
34  setProtectedArea("Methods");
35  @}
```

LINE 3: swapping to script mode: the value of *this.name* is put into the output file, knowing that the variable *this* is determined by the second parameter that is passed to the procedure `generate` (see section 4.3.85 and below). If the notation appears confusing to you (where does the writing mode ends, where does the script mode starts or the contrary), you can choose to inlay the variables in tags **'<%'** and **'%>'**.

LINE 4: swapping once again to script mode for writing the inheritance, if any

LINE 7: swapping to text mode,

LINE 10: we'll need a function to convert a type specifier of the tiny modeling language to JAVA, which expects the attribute's node (parameter mode is `variable`, instead of `value`),

LINE 13: we have chosen `java.util.ArrayList` to represent an array, why not?

LINE 18: swapping to script mode for declaring the attributes of the class

LINE 22: swapping to text mode for putting the constructor into the output file,

LINE 29: swapping to script mode for implementing the accessors to the attributes of the class

LINE 30: the predefined function `toUpperString` capitalizes the parameter,

LINE 34: the procedure `setProtectedArea` (see section 4.6.34) adds a protected area that is intended to the user and that is preserved during a generation process,

LINE 35: swapping to text mode for writing the trailing brace,

The *leader script* must be changed to require the generation of each class in JAVA:

CODEWORKER *command line to execute:*
```
-script Scripts/Tutorial/GettingStarted/Tiny-leaderScript1.cws
```

```
    // file "Scripts/Tutorial/GettingStarted/Tiny-leaderScript1.cws":
1  parseAsBNF("Scripts/Tutorial/GettingStarted/Tiny-BNFparsing1.cwp",
```

```
project, "Scripts/Tutorial/GettingStarted/Tiny.tml");
2
3 foreach i in project.listOfClasses {
4     generate("Scripts/Tutorial/GettingStarted/Tiny-JAVA.cwt", i,
"Scripts/Tutorial/GettingStarted/tiny/" + i.name + ".java");
5 }
6
```

LINE 4: the second argument is waiting for a tree node that will be accessed into the *pattern script* via the predefined variable `this`, which has been encountered above,

Output:

```
this file has been parsed successfully
```

Let have a look to the following generated file:

```
    // file "Scripts/Tutorial/GettingStarted/tiny/D.java":
package tiny;

public class D {
    // attributes:
    private A _a = null;
    private java.util.ArrayList/*<C>*/ _c = null;

    //constructor:
    public D() {
    }

    // accessors:
    public A getA() { return _a; }
    public void setA(A a) { _a = a; }
    public java.util.ArrayList/*<C>*/ getC() { return _c; }
    public void setC(java.util.ArrayList/*<C>*/ c) { _c = c; }
//##protect##"Methods"
//##protect##"Methods"
}
```

2.6   Expanding text with a pattern script

We'll learn about another mode of generation: expanding a file. Let suppose that you want to inlay generated code into an existing file. The way to do it is first to insert a special comment at the expected place. This comment begins with **##markup##** and is followed by a sequence of characters written between double quotes and called the *markup key*.

Here is a little HTML file that is going to be expanded:

```
    // file "Scripts/Tutorial/GettingStarted/Tiny.html":
<HTML>
    <HEAD>
    </HEAD>
    <BODY>
<!-##markup##"classes"->
```

```
            </BODY>
        </HTML>
```

The markup key is called *"classes"* and is put into the file like it: `<!- -##markup##"classes"-->`.

Now, we'll implement a short script that is intended to populate the markup area with all classes of the design, displayed into tables:

```
       // file "Scripts/Tutorial/GettingStarted/Tiny-HTML.cwt":
 1  @
 2  if getMarkupKey() == "classes" {
 3      foreach i in project.listOfClasses {
 4          @ <TABLE>
 5              <TR>
 6                  <TD colspan=3><B>@i.name@</B></TD>
 7              </TR>
 8              <TR>
 9                  <TD><EM>Attribute</EM></TD><TD><EM>Type</EM></TD>
<TD><EM>Description</EM></TD>
10              </TR>
11  @
12          foreach j in i.listOfAttributes {
13              @ <TR>
14                  <TD><I>@j.name@</I></TD><TD><CODE>@
15              @@j.class.name@@
16              if j.isArray {
17                  @[]@
18              }
19              @</CODE></TD><TD>@
20              setProtectedArea(i.name + "::" + j.name);
21              @</TD>
22              </TR>
23  @
24          }
25          @ </TABLE>
26  @
27      }
28  }
```

LINE 2: the function `getMarkupKey()` returns the current expanding markup that is handled,
LINE 3: all classes will be presented sequentially into tables of 3 columns, whose title is the name of the class, and rows are populated with attributes,
LINE 12: the *name*, *Type* and *Description* of all attributes of the class are presented into the table,
LINE 15: the type is expressed in the syntax of our tiny modeling language,
LINE 20: the description of an attribute must be filled by the user into a protected area, so as to preserve it from an expansion to another,

The *leader script* has to take into account the expansion of the HTML file:

CODEWORKER *command line to execute:*
`-script Scripts/Tutorial/GettingStarted/Tiny-leaderScript2.cws`

```
       // file "Scripts/Tutorial/GettingStarted/Tiny-leaderScript2.cws":
```

```
   1 parseAsBNF("Scripts/Tutorial/GettingStarted/Tiny-BNFparsing1.cwp",
project, "Scripts/Tutorial/GettingStarted/Tiny.tml");
   2
   3 foreach i in project.listOfClasses {
   4     generate("Scripts/Tutorial/GettingStarted/Tiny-JAVA.cwt", i,
"Scripts/Tutorial/GettingStarted/tiny/" + i.name + ".java");
   5 }
   6
   7 traceLine("expanding file 'Tiny0.html'...");
   8 setCommentBegin("<!-");
   9 setCommentEnd("->");
  10 expand("Scripts/Tutorial/GettingStarted/Tiny-HTML.cwt",
project, "Scripts/Tutorial/GettingStarted/Tiny0.html");
  11 //normal;
```

LINE 8: to expand a file, the interpreter has to know the format of comments used for declaring the markups. If the format isn't correct, the file will not be expanded.

LINE 10: be careful to call the procedure `expand()` and not to confuse with `generate()`! Remember that a classic generation rewrites all according to the directives of the *pattern script* and preserves protected areas, but doesn't recognize markup keys.

Output:

```
this file has been parsed successfully
expanding file 'Tiny0.html'...
```

It hasn't a great interest to present here the content of the HTML once it has been expanded, but you can display it (file *"Scripts/Tutorial/GettingStarted/Tiny0.html"*) into your browser. You'll notice into the source code that the expanded text is put between tags **<!- -##begin##"classes"- ->** and **<!- -##end##"classes"- ->**. Don't type text into this tagged part, except into protected areas, because the next expansion will destroy the tagged part.

For discovering more about CODEWORKER through a more complex example, please read the next chapter. You'll learn how to do translations from a format to another, and to use template functions or BNF clauses (very efficient for readability and extension!), and a lot of various things. But it is recommended to practice a little before.

Discovering more with an example

The first time, we recommend to read the precedent chapter, more approachable, before reading this one.

Let imagine that we dispose of a design expressed in a simple modeling language, like it:

```
     // file "GettingStarted/SolarSystem0.sml":
1 class Planet {
2     double diameter;
3     double getDistanceToSun(int day, int month, int year);
4 }
5
6 class Earth :  Planet {
7     string[] countryNames;
8 }
9
10 class SolarSystem {
11     aggregate Planet[] planets;
12 }
```

LINE 1: a class is declared with keyword **class**
LINE 2: declaration of attributes in a syntax close to C++ or JAVA
LINE 3: declaration of methods in a syntax close to C++ or JAVA
LINE 6: a class may inherit from an other ; the syntax looks like C++, see **':'**
LINE 7: an attribute may be an array ; the syntax looks like JAVA
LINE 11: an attribute may be an object or an array of objects, and an object may be an aggregation (meaning that it belongs to the instance),

This simple modeling language conforms to a BNF grammar (see paragraph 4.3.216 to obtain information about the elements of a BNF syntax):
world ::= [*class_declaration*]*
class_declaration ::= **"class"** *IDENT* ['**:**' *IDENT*]? *class_body*
class_body ::= '**{**' [*attribute_decl* | *method_decl*]* '**}**'
attribute_decl ::= *type_specifier IDENT* ';'
method_decl ::= *type_specifier IDENT* '**(**' [*parameters_decl*]? '**)**'
';'
parameters_decl ::= *parameter* [',' *parameters_decl*]*
parameter ::= [*parameter_mode*]? *type_specifier IDENT*
parameter_mode ::= **"in"** | **"inout"** | **"out"**
type_specifier ::= *basic_type* ['**[**' '**]**']?
basic_type ::= **"int"** | **"double"** | **"string"** |
"boolean" | *class_specifier*

```
class_specifier ::= ["aggregate"]? IDENT
IDENT ::= ['a'..'z'|'A'..'Z'|'_'] ['a'..'z'|'A'..'Z'|'_'|'0'..'9']*
```

Starting from the desing file "SolarSystem0.sml" seen before, which conforms to the *Simple Modeling Language* described just above, we propose to implement the source code for classes and a light documentation.

3.1   The parse tree

CODEWORKER doesn't belong to the category of typed languages. It recognizes only the *tree* as structured type and the *string* as basic type (that may however represent an integer or a boolean, ...). Each node may contain a string as a value, and/or an array of nodes. The main tree is called `project`, which is the name of its root node, accessible everywhere into scripts.

Now, the best way to understand how to handle the tree is to run the console, and to practice some examples.

Type `CodeWorker` to the shell to set the console mode. A cursor is waiting for your commands.

Type `set a = "little";` and press enter. Don't forget the semi-colon at the end of the line. If absent, the console wait for more input: type the expected semi-colon, and it should be right.

What is the impact of the line you typed? You assigned `"little"` to the variable a, which doesn't exist. So, a node named 'a' has been added into the main parse tree (called `project`, remember), to which the variable a points. You noticed that a varning has occurred. It means that you assigned a value to a node that doesn't exist yet. In fact, the instruction `set` supposes that the variable to assign already exists, and a warning has been thrown to prevent you of a spelling error (perhaps do you intended to type another variable that already exists?) or a logic mistake (at this point of the program, the variable should exist, so what?). It is important to offer this protection, because the language isn't typed, and so, a lot of errors may be reported during the runtime.

The variable a has been added, even if the warning has occurred, but we prefer the instruction `insert` to add a new node properly : type `insert b = "big";` and press enter. No warning was displayed. Now, the root `project` node contains two sub-nodes, called 'a' and 'b', and we control it by typing `traceObject(project);`. The following lines are displayed:

```
    Tracing variable 'project':
            a = "little"
            b = "big"
    End of variable's trace 'project'.
```

Let's go further. What about storing a list of items?
Type `insert classes["Planet"].name = "Planet";`. A node node called 'classes' has been added to `project`, and then an array entry called `"Planet"` has been pushed. This entry points to a node, to which 'name' is added, and node 'name' is worth `"Planet"`.

Type `insert classes["Earth"].name = "Earth";` and then ask for tracing node 'project'. The following lines are displayed:

```
    Tracing variable 'project':
           a = "little"
           b = "big"
           classes = ""
           classes["Planet", "Earth"]
    End of variable's trace 'project'.
```

Notice that the node 'classes' has no value (but could have!) and contains an array of nodes where entries are "Planet" and "Earth".

To iterate items of array 'classes', type `foreach i in classes traceLine("handling class '" + i.name + "'...");` and see the result:

```
    handling class 'Planet'...
    handling class 'Earth'...
```

Variable 'i' is an iterator and is declared locally for processing the `foreach` instruction. We'll see further that the statement `local` allows declaring a tree to the stack.

What you know about the parse tree in CODEWORKER is sufficient to tackle the next section.

3.2   Parsing our design

CODEWORKER provides two different approaches for parsing files.

3.2.1   The parsing scripts that read tokens

Those that aren't familiar with a BNF representation will perhaps be more self-assured in using a procedure-driven parsing, where control resides within the implementation and where all tokens are explicitly read by a devoted operation. But it means for instance that ignoring blanks and comments must be indicated explicitly between reading of tokens.

The parsing scripts that read tokens are the oldest way to parse into CODEWORKER and are the fastest mode too. But it doesn't offer the same flexibility as BNF scripts, which are syntax-oriented.

Below is an example of what a script that reads tokens looks like:

```
      // file "GettingStarted/SimpleML-token-reading.cws":
 1  declare function readType();
 2
 3  while skipEmptyCpp() {
 4      if !readIfEqualToIdentifier("class") error("'class'
expected");
 5      skipEmptyCpp();
 6      local sClassName = readIdentifier();
 7      if !sClassName error("class name expected");
 8      skipEmptyCpp();
 9      if readIfEqualTo(":") {
10          skipEmptyCpp();
11          local sParentName = readIdentifier();
12          if !sParentName error("parent name expected for class '"
+ sClassName + "'");
```

```
13          skipEmptyCpp();
14       }
15       if !readIfEqualTo("{") error("'{' expected");
16       skipEmptyCpp();
17       while !readIfEqualTo("}") {
18           skipEmptyCpp();
19           readType();
20           skipEmptyCpp();
21           local sMemberName = readIdentifier();
22           if !sMemberName error("attribute or method name
expected");
23           skipEmptyCpp();
24           if readIfEqualTo("(") {
25               skipEmptyCpp();
26               if !readIfEqualTo(")") {
27                   do {
28                       skipEmptyCpp();
29                       local iPosition = getInputLocation();
30                       local sMode = readIdentifier();
31                       if !sMode error("parameter type or mode
expected");
32                       if (sMode != "in") && (sMode != "out") &&
(sMode != "inout") {
33                           setInputLocation(iPosition);
34                           set sMode = "";
35                       }
36                       skipEmptyCpp();
37                       readType();
38                       skipEmptyCpp();
39                       local sParameterName = readIdentifier();
40                       if !sParameterName error("parameter name
expected");
41                       skipEmptyCpp();
42                   } while readIfEqualTo(",");
43                   if !readIfEqualTo(")") error("')' expected");
44               }
45               skipEmptyCpp();
46           }
47           if !readIfEqualTo(";") {
48               error("';' expected to close an attribute, instead of
'" + readChar() + "'");
49           }
50           skipEmptyCpp();
51       }
52    }
53    traceLine("the file has been read successfully");
54
55    function readType() {
56       local sType = readIdentifier();
57       if !sType error("type modifier or name expected, instead of
'" + readChar() + "'");
```

```
58      if sType == "aggregate" {
59          skipEmptyCpp();
60          sType = readIdentifier();
61          if !sType error("aggregated class name expected");
62      }
63      skipEmptyCpp();
64      if readIfEqualTo("[") {
65          skipEmptyCpp();
66          if !readIfEqualTo("]") error("']' expected to close an
array declaration");
67      }
68  }
```

LINE 1: forward declaration of method `readType()`, so as to start explanations about how to implement BNF clause *world ::= [class_declaration]**,

LINE 3: do a loop while the end of file hasn't been reached, skipping blanks and C++ comments: `skipEmptyCpp()` returns `false` only if an error occurs while reading the stream or the file has completed,

LINE 4: waiting for token `"class"` as an identifier (doesn't accept `"class"` as the beginning of another identifier, such as `"classes"`). If not found, an error occurs. This token announces a class declaration.

LINE 5: a disadvantage of writing a procedure-driven reading/parsing: don't forget to skip explicitly blanks and comments by yourself,

LINE 6: populates a local variable with an identifier token that represents the name of the class

LINE 7: if an identifier token hasn't been found (token is empty), an error is thrown,

LINE 9: if the file location points to `":"`, announcing the inheritance, function `readIfEqualTo(":")` returns `true`, and the location moves after the matched expression. If it fails, the file location remains the same.

LINE 15: body of the class declaration expected

LINE 17: while inside the class body, reading of attribute and method members,

LINE 19: we don't conform exactly to the BNF: beginning of method and attribute declaration is factorized,

LINE 21: name of the attribute or method member,

LINE 24: not any more ambiguity : it starts by a parenthesis when the members is a method,

LINE 27: the method expects at least one parameter,

LINE 29: we keep the current file position, to be able to come back if the next token isn't an access mode (`"in"`, `"out"` or `"inout"`),

LINE 33: we were reading a basic type, instead of a parameter access mode: we come back to the beginning of this token and the mode is set as empty (no mode). Of course, it is possible not to waste time like this, and to optimize function `readType()` by passing the token as a parameter. But here is the occasion of discovering how to handle the file position.

LINE 37: type of the current parameter is expected,

LINE 39: name of the current parameter is expected,

LINE 42: parameters are separated by commas,

LINE 47: both attributes and methods must finish with a semi colon,

LINE 48: function `readChar()` reads just one character, or returns an empty string if the end of file has been reached,

LINE 53: once the read of file has completed, a message of success is written,

LINE 55: user-defined function ; may return a value or not. The declaration always starts with keyword `function`, even if it announces a procedure (no return value). Reading a type is called at several points of the grammar, so the code is factorized in the procedure `readType()`. It doesn't return any value about success or failure, because an error is thrown in case of syntax mismatch.

LINE 58: does the keyword is a modifier? If not `sType` contains a basic type or a class name
LINE 60: reads the name of the aggregated class
LINE 64: perhaps that the type is an array, represented by `[]`,

This script seems quite far from the BNF of our simple modeling language, while it implements it in a procedural way. It is able to read a well-formed design file, as our solar system presented at the beginning of the chapter. It doesn't care about populating a parse tree yet, but produces contextual error messages when the design file doesn't conform to the BNF.

Let apply the script on the design file:

```
parseFree("GettingStarted/SimpleML-token-reading.cws",
project, "GettingStarted/SolarSystem0.sml");
```

Output:

`the file has been read successfully`

Now, let improve the script to allow populating a parse tree:

```
    // file "GettingStarted/SimpleML-token-parsing.cws":
1 declare function readType(myType :  node);
2
3 while skipEmptyCpp() {
4     if !readIfEqualToIdentifier("class") error("'class'
expected");
5     skipEmptyCpp();
6     local sClassName = readIdentifier();
7     if !sClassName error("class name expected");
8     insert project.listOfClasses[sClassName].name = sClassName;
9     skipEmptyCpp();
10    if readIfEqualTo(":") {
11        skipEmptyCpp();
12        local sParentName = readIdentifier();
13        if !sParentName error("parent name expected for class '"
+ sClassName + "'");
14        insert project.listOfClasses[sClassName].parent =
sParentName;
15        skipEmptyCpp();
16    }
17    if !readIfEqualTo("{") error("'{' expected");
18    skipEmptyCpp();
19    local myClass;
20    ref myClass = project.listOfClasses[sClassName];
21    while !readIfEqualTo("}") {
22        skipEmptyCpp();
23        local myType;
24        readType(myType);
25        skipEmptyCpp();
26        local sMemberName = readIdentifier();
27        if !sMemberName error("attribute or method name
expected");
28        skipEmptyCpp();
```

```
29          if readIfEqualTo("(") {
30              insert myClass.listOfMethods[sMemberName].name =
sMemberName;
31              if myType.name != "void" {
32                  setall myClass.listOfMethods[sMemberName].type =
myType;
33              }
34              skipEmptyCpp();
35              if !readIfEqualTo(")") {
36                  local myMethod;
37                  ref myMethod = myClass.listOfMethods[sMemberName];
38                  do {
39                      skipEmptyCpp();
40                      local iPosition = getInputLocation();
41                      local sMode = readIdentifier();
42                      if !sMode error("parameter type or mode
expected");
43                      if (sMode != "in") && (sMode != "out") &&
(sMode != "inout") {
44                          setInputLocation(iPosition);
45                          set sMode = "";
46                      }
47                      skipEmptyCpp();
48                      local myParameterType;
49                      readType(myParameterType);
50                      skipEmptyCpp();
51                      local sParameterName = readIdentifier();
52                      if !sParameterName error("parameter name
expected");
53                      insert myMethod.listOfParameters[sParameterName].name
= sParameterName;
54                      setall myMethod.listOfParameters[sParameterName].type
= myParameterType;
55                      if sMode {
56                          insert myMethod.listOfParameters[sParameterName].name
= sMode;
57                      }
58                      skipEmptyCpp();
59                  } while readIfEqualTo(",");
60                  if !readIfEqualTo(")") error("')' expected");
61              }
62              skipEmptyCpp();
63          } else {
64              insert myClass.listOfAttributes[sMemberName].name =
sMemberName;
65              setall myClass.listOfAttributes[sMemberName].type =
myType;
66          }
67          if !readIfEqualTo(";") error("';' expected to close an
attribute, instead of '" + readChar() + "'");
68          skipEmptyCpp();
```

```
69        }
70    }
71    traceLine("the file has been parsed successfully");
72
73    function readType(myType :  node) {
74        local sType = readIdentifier();
75        if !sType error("type modifier or name expected, instead of
'" + readChar() + "'");
76        if sType == "aggregate" {
77            insert myType.isAggregation = true;
78            skipEmptyCpp();
79            sType = readIdentifier();
80            if !sType error("aggregated class name expected");
81        }
82        insert myType.name = sType;
83        if (sType != "int") && (sType != "double") && (sType !=
"boolean") && (sType != "string") {
84            insert myType.isObject = true;
85        }
86        skipEmptyCpp();
87        if readIfEqualTo("[") {
88            skipEmptyCpp();
89            if !readIfEqualTo("]") error("']' expected to close an
array declaration");
90            insert myType.isArray = true;
91        }
92    }
```

LINE 8: about parsing, classes are modeled into node **project.listOfClasses[***sClassName***]**. Its attribute `name` contains the value of *sClassName*.

LINE 14: this class inherits from a parent, so the optional attribute `parent` of the class is populated with the value of *sParentName*,

LINE 19: to work easier with the current class node `project.listOfClasses[sClassName]`, we define a reference to it, called `myClass`,

LINE 23: the class is populated with the characteristics of the member once its declaration has finished. Otherwise, it may confuse between an attribute or a method declaration. So, we should have factorized the type declaration and the name of the member into a common clause, for example.

LINE 30: about parsing, methods are modeled into node **myClass.listOfMethods[***sMemberName***]**,

LINE 31: attribute `name` is compulsory into a *type* node, so if `myType.name` returns `"void"`, there is no return type,

LINE 36: to work easier with the current class node `myClass.listOfMethods[sMemberName]`, we define a reference to it, called `myMethod`,

LINE   53:   about   parsing,   parameters   are   modeled   into   node **myMethod.listOfParameters[***sParameterName***]**,

LINE 64: about parsing, attributes are modeled into node **myClass.listOfAttributes[***sMemberName***]**,

LINE 65: the type is allocated on the stack, so it is copied into branch `type` (no node reference) integrally,

LINE 71: once the parsing of file has achieved, a message of success is written,

LINE 73: function `readType()` requires a node into which description of type will be populated,

LINE 77: about parsing, **myType.isAggregation** contains `true` if type is an array,

LINE 82: about parsing, **myType.name** contains the name of basic type,

LINE 83: check whether the type is a basic one or a class specifier,

LINE 84: about parsing, **myType.isObject** contains `true` because we suppose that this type is a class specifier (by default: it isn't a basic type),
LINE 90: about parsing, **myType.isArray** contains `true` if type is an array,

The first version of the script was just able to read a well-formed design file written in the simple modeling language. The second version validates the file and populates the parse tree:

```
parseFree("GettingStarted/SimpleML-token-parsing.cws",
project, "GettingStarted/SolarSystem0.sml");
```

Output:

```
the file has been parsed successfully
```

3.2.2   The parsing scripts that describe a BNF syntax

A BNF is more flexible and more synthetic than a procedural description of parsing. CODEWORKER accepts parsing scripts that conform to a BNF.

For more information about elements of syntax for a BNF, let have a look to paragraph 4.3.216.

Below is an example of what a BNF script looks like:

```
    // file "GettingStarted/SimpleML-reading.cwp":
 1 // syntactical clauses:
 2 world ::= #ignore(C++) [class_declaration]* #empty
 3           => { traceLine("file read successfully"); };
 4 class_declaration ::= IDENT:"class" IDENT [':'  IDENT]?
class_body;
 5 class_body ::= '{' [attribute_decl | method_decl]* '}';
 6 attribute_decl ::= type_specifier IDENT ';';
 7 method_decl ::= [IDENT:"void" | type_specifier] IDENT
 8               '(' [parameters_decl]?  ')'  ';';
 9 parameters_decl ::= parameter [',' parameters_decl]*;
 10 parameter ::= [parameter_mode]?  type_specifier IDENT;
 11 parameter_mode ::= IDENT:{"in", "inout", "out"};
 12 type_specifier ::= basic_type ['[' ']']?;
 13 basic_type ::= "int" | "boolean" | "double" | "string" |
class_specifier;
 14 class_specifier ::= ["aggregate"]?  IDENT;
 15
 16 // lexical clauses:
 17 IDENT ::= #!ignore ['a'..'z'|'A'..'Z'|'_']
 18                    ['a'..'z'|'A'..'Z'|'_'|'0'..'9']*;
```

LINE 2: the world to model is composed of classes ; some special commands are used:

- `#ignore(C++)` means that blank characters and *C++-like* comments will be ignored between pattern matching instructions,

- `#empty` means that the position must point to the end of the input file,

- `=> traceLine("file read successfully");` means that a trace must be executed

---

just after matching with the end of file (the pattern matching instruction is `#empty`) ; let retain that an instruction or a block of instructions is announced by **'=>'**,

LINE 4: a class declaration begins with identifier "class", and *IDENT*:`"class"` means that an identifier is expected, and that this identifier is worth `"class"`. This instruction isn't identical to `"class"` *IDENT* that validates the expression *"classes"*, where *IDENT* matches to *"es"*. A class has a name, read by the first *IDENT* clause call, and may inherit from a parent, read by the second *IDENT*

LINE 5: the body of a class is composed of attributes and methods

LINE 6: the attribute is preceded by its type, and *IDENT* reads the name of the attribute

LINE 7: the method has a return type or expects `void` keyword, and may expect some parameters ; *IDENT* reads the name of the method

LINE 9: a comma separates parameters

LINE 10: an access mode may be specified to the parameter ; the type is then specified, and *IDENT* reads the name

LINE 11: a parameter may be passed:

- **in** and its value cannot be changed by the method,

- **inout** and its value may be changed into the method,

- **out** and the method doesn't care about the initial value of the parameter, but is expected to assign a value to it into the body,

The pattern *IDENT*:`{"in", "inout", "out"}` means that the identifier must match with one of the constant strings listed between brackets. It isn't identical to the pattern `"in"` | `"inout"` | `"out"` that validates the beginning of *"int"*.

LINE 12: a type is a *basic type* or an array of basic types

LINE 13: some basic types, including object types

LINE 14: *IDENT* reads the class name, and the object may be aggregated

LINE 17: this clause reads an identifier, such as *pretty_pig1* ; `#!ignore` means that no character is ignored, even if it matches C++ comment or a blank. If we forget clause `#!ignore`, then *IDENT* will validate *pretty/*comment*/_pig 1* as an identifier.

This BNF script is very close to the BNF of our simple modeling language, and is able to read a well-formed design file, as our solar system presented at the beginning of the chapter. It doesn't care about populating a parse tree yet, and doesn't produce a contextual error message when the design file doesn't conform to the BNF.

Let apply the BNF script on the design file:

```
parseAsBNF("GettingStarted/SimpleML-reading.cwp",
project, "GettingStarted/SolarSystem0.sml");
```

Output:

```
file read successfully
```

About differences, note that each BNF rule must end with a semi colon, and that they have to indicate what is their behaviour while encountering blanks and comments.

Now, let improve the BNF script to allow populating a parse tree, or throwing an error when a syntax error has occurred:

```
    // file "GettingStarted/SimpleML-parsing.cwp":
 1 // syntactical clauses:
```

```
 2  world ::= #ignore(C++) [class_declaration]* #empty
 3          => {
 4              traceLine("file parsed successfully");
 5              saveProject("Scripts/Tutorial/SolarSystem0.xml");
 6          };
 7  class_declaration ::= IDENT:"class" #continue
 8          IDENT:sClassName
 9              => insert project.listOfClasses[sClassName].name
= sClassName;
10          [':'  #continue IDENT:sParentName
11              => insert project.listOfClasses[sClassName].parent
= sParentName;
12          ]?
13          class_body(project.listOfClasses[sClassName]);
14  class_body(myClass :  node) ::= '{'
15      [attribute_decl(myClass) | method_decl(myClass)]* '}';
16  attribute_decl(myClass :  node) ::=
17          => local myType;
18          type_specifier(myType) IDENT:sAttributeName ';'
19          => {
20              insert myClass.listOfAttributes[sAttributeName].name
= sAttributeName;
21              setall myClass.listOfAttributes[sAttributeName].type
= myType;
22          };
23  method_decl(myClass :  node) ::=
24          => local myType;
25          [IDENT:"void" | type_specifier(myType)]
26          IDENT:sMethodName '('
27          #continue
28              => {
29                  insert myClass.listOfMethods[sMethodName].name
= sMethodName;
30                  if myType.name
31                      setall myClass.listOfMethods[sMethodName].type
= myType;
32              }
33          [parameters_decl(myClass.listOfMethods[sMethodName])]?
')'  ';';
34  parameters_decl(myMethod :  node) ::=
35              parameter(myMethod)
36          [',' #continue parameters_decl(myMethod)]*;
37  parameter(myMethod :   node) ::=
38          [parameter_mode]?:sMode
39          => local myType;
40          type_specifier(myType)
41          IDENT:sParameterName
42              => {
43                  insert myMethod.listOfParameters[sParameterName].name
= sParameterName;
44                  setall myMethod.listOfParameters[sParameterName].type
```

```
= myType;
45                   if sMode {
46                       insert myMethod.listOfParameters[sParameterName].name
= sMode;
47                   }
48               };
49 parameter_mode ::= IDENT:{"in", "inout", "out"};
50 type_specifier(myType :  node) ::=
51     basic_type(myType)
52     ['[' #continue ']' => insert myType.isArray = true; ]?;
53 basic_type(myType :  node) ::=
54     ["int" | "boolean" | "double" | "string"]:myType.name
55         |
56     class_specifier(myType);
57 class_specifier(myType :  node) ::=
58     ["aggregate" => insert myType.isAggregation = true; ]?
59     IDENT:myType.name => {insert myType.isObject = true; };
60
61 IDENT ::= #!ignore ['a'..'z'|'A'..'Z'|'_']
62                    ['a'..'z'|'A'..'Z'|'_'|'0'..'9']*;
```

LINE 2: the pattern *[class_declaration]** always matches with the parsed file, so the rule will continue in sequence in any case (supposing that no error has occurred into clause *class_declaration*) and the end of file will be checked. If not reached, it doesn't write the message `"file read successfully"`,

LINE 7: once keyword `"class"` has been matched, there is no ambiguity : we are handling a class declaration and the rule must continue in sequence. To require that, instruction `#continue` is written after pattern `"class"`. If a pattern of the sequence doesn't match the parsed file, the parser throws a syntax error automatically.

LINE 8: the identifier that matches with clause call *IDENT* is assigned to the local variable `sClassName` : on contrary of other types of script, a new variable is considered as local, instead of an new attribute added to the current node **this**,

LINE 9: about parsing, classes are modeled into node **project.listOfClasses[***sClassName***]**. Its attribute `name` contains the value of *sClassName*.

LINE 10: if the class inherits from a parent, `':'` is necessary followed by an identifier (pattern `#continue`), and the identifier that matches with clause call *IDENT* is assigned to the local variable `sClassName`,

LINE 11: this class inherits from a parent, so the optional attribute `parent` of the class is populated with the value of *sParentName*,

LINE 14: clause `class_body` expects an argument: the class node into which the class members must be described (`myClass :` **node**),

LINE 16: the class is populated with the characteristics of the attribute once its declaration has finished. Otherwise, it may confuse with the beginning of a method declaration. To avoid this ambiguity, we should have factorized the type declaration and the name of the member into a common clause, for example.

LINE 20: about parsing, attributes are modeled into node **myClass.listOfAttributes[***sAttributeName***]**,

LINE 21: the type is allocated on the stack, so it is copied into branch `type` (no node reference) integrally,

LINE 23: the class is populated with the characteristics of the method once the opened parenthesis is recognized,

LINE 27: from here, there is no doubt that we are parsing a method declaration,

LINE 29: about parsing, methods are modeled into node **myClass.listOfMethods[***sMethodName***]**,

LINE 30: attribute `name` is compulsory into a *type* node, so if condition `myType.name` returns

`false`, there is no return type (`void`),

LINE 36: a parameter declaration is expected after the comma,

LINE 43: about parsing, parameters are modeled into node **myMethod.listOfParameters[*sParameterName*]**,

LINE 52: about parsing, **myType.isArray** contains `true` if type is an array,

LINE 54: about parsing, **myType.name** contains the name of basic type,

LINE 58: about parsing, **myType.isAggregation** contains `true` if the object is aggregated,

LINE 59: about parsing, **myType.isObject** contains `true` because this type is a class specifier,

LINE 61: the lexical clause *IDENT* recognizes identifiers and might be replaced by the predefined clause `#readIdentifier`, which does the same work,

The first version of the script was just able to read a well-formed design file written in the simple modeling language. The second version validates the file and populates the parse tree:

```
parseAsBNF("GettingStarted/SimpleML-parsing.cwp",
project, "GettingStarted/SolarSystem0.sml");
```

Output:

```
file parsed successfully
```

3.3   Decorating the parse tree

Once our design file has been parsed (either procedure-driven or BNF-driven, we don't care), there is sometimes a little more work to acomplish on the parse tree. It may be verifying consistency of the whole, as checking existence of each class referenced as association or parent. It may also be reorganizing the graph differently, so as to simplify tasks of source code generation. We call it *decorating the parse tree* in the CODEWORKER vocabulary.

The next script proposes to check the existence of each class specifier types and to keep a reference to the node that describes this class specifier. Some nodes change their nature (`myClass.parent` becomes a reference to the parent node, for example), some other are added (for object types, the new node `myType.class` keeps a reference to the class):

```
     // file "GettingStarted/TreeDecoration.cws":
 1 foreach myClass in project.listOfClasses {
 2    if myClass.parent {
 3       if !findElement(myClass.parent, project.listOfClasses)
 4          error("class '" + myClass.parent + "' doesn't exist
while class '"
 5                + myClass.name + "intends to inherit from it");
 6       ref myClass.parent = project.listOfClasses[myClass.parent];
 7    }
 8    foreach myAttribute in myClass.listOfAttributes {
 9       local myType;
10       ref myType = myAttribute.type;
11       if myType.isObject {
12          if !findElement(myType.name, project.listOfClasses)
13             error("class '" + myType.name + "' doesn't exist
while attribute '"
14                + myClass.name + "::" + myAttribute.name +
"' refers to it");
```

```
15            ref myType.class = project.listOfClasses[myType.name];
16        }
17    }
18    foreach myMethod in myClass.listOfMethods {
19        if existVariable(myMethod.type) && myMethod.type.isObject
{
20            localref myType = myMethod.type;
21            if !findElement(myType.name, project.listOfClasses)
22                error("class '" + myType.name + "' doesn't exist
while method '"
23                      + myClass.name + "::" + myMethod.name + "'
refers to it");
24            ref myType.class = project.listOfClasses[myType.name];
25        }
26        foreach myParameter in myMethod.listOfParameters {
27            localref myType = myParameter.type;
28            if myType.isObject {
29                if !findElement(myType.name, project.listOfClasses)
30                    error("class '" + myType.name
31                          + "' doesn't exist while method '"
32                          + myClass.name + "::" + myMethod.name
33                          + "' refers to it");
34                ref myType.class = project.listOfClasses[myType.name];
35            }
36        }
37    }
38 }
```

LINE 1: we iterate all classes,

LINE 2: if field `parent` is filled, we check its existence and then, we change it as a reference to the parent class,

LINE 8: we iterate all attributes of each class,

LINE 11: only object attributes are interesting,

LINE 12: check whether the class exists or not into the array node that contains all classes: does the key `myType.name` exist as an array entry of node `project.listOfClasses`?

LINE 15: to optimize navigating into the parse tree later, we keep a reference to the class into new node **myType.class**,

LINE 18: we iterate all methods of each class,

LINE 26: we iterate all parameters of each method,

Now, we dispose of a parsing script that loads well-formed *Simple-Modeling* designs, and a script that decorates the parse tree. It is time to write a *leader script* that will take in charge calling tasks of parsing, tree decoration and source code generation:

CODEWORKER *command line to execute:*
```
-I Scripts/Tutorial/GettingStarted -define DESIGN_FILE=SolarSystem0.sml
-script LeaderScript0.cws
```

```
    // file "GettingStarted/LeaderScript0.cws":
1 if !getProperty("DESIGN_FILE")
2    error("'-define DESIGN_FILE=file' expected on the command
line");
3 traceLine("'Simple Modeling' design file to parse = \""
```

```
4              + getProperty("DESIGN_FILE") + "\"");
5 parseAsBNF("SimpleML-parsing.cwp",
6          project, getProperty("DESIGN_FILE"));
7 #include "TreeDecoration.cws"
```

LINE 1: we expect the design as a file that conforms to our *Simple-Modeling Language* ; the file name is given to the definition preprocessor DESIGN_FILE on the command line by typing -define DESIGN_FILE=SolarSystem0.sml,
LINE 5: the file is parsed thanks to our previous BNF script,
LINE 7: the source code for decorating tree is included here, and its content will be executed just after the parsing,

3.4   Generating code

A script that is intended to source code generation is called a *pattern script* in the CODEWORKER vocabulary. The output file is rewritten completely after the protected areas of user's source code have been preserved.

Such a script begins with a sequence of characters exactly like they must be written into the output file, up to it encounters special character **'@'** or JSP-like tag **'<%'**. Then it swaps into script mode, and everything is interpreted as script instructions, up to special character **'@'** or JSP-like tag **'%>'** are encountered. Content of the script file is again understood as a sequence of characters to write into the output file, up to the next special character. And it continues swapping from a mode to another...

For convenience, the script mode may be just restrained to an expression (often the name of a variable) whose value is written into the output file.

To do source code generation, we'll need some useful functions, such as converting a *Simple-Modeling* type to its C++ representation. These functions might be included into the leader script, so as to be shared by all *pattern* scripts.

We'll discover a new type of functions, called *template functions* that bring a little generic programming in the language: let imagine that we need function getType(myType :   node), to decline for every language we could have to generate (C++ and JAVA in this chapter). You plan to generate an object library from the design you have written in the *Simple Modeling Language*. This object library will be delivered both in C++ and JAVA, and a technical documentation will come with each of these implementations. This technical documentation will give the signature of methods and the type of attributes in the language the developer will choose. So the C++ documentation will be slightly different from the JAVA one, just at the level of type's spelling. Normally, you'll write the following lines to recover the type depending on the language for which you are producing the documentation:

```
if doc_language == "C++" {
sType = getCppType(myParameterType);
} else if doc_language == "JAVA" {
sType = getJAVAType(myParameterType);
} else {
error("unrecognized language '" + doc_language + "'");
}
```

Thanks to the template functions, you may replace the precedent lines by the next one:

```
sType = getType<doc_language>(myParameterType);
...
function getType<"JAVA">(myType : node) {
... // implementation for returning a Java type
}

function getType<"C++">(myType : node) {
... // implementation for returning a C++ type
}
```

During the execution, the function getType<**T**>(myType : node) resolves on what instantiated function it has to dispatch: either getType<**"JAVA"**>(myType : node) or getType<**"C++"**>(myType : node), depending on what value is assigned to variable *doc_language*.

Trying to call an instantiated function that doesn't exist, raises an error at runtime. However, one might imagine an implementation by default. For instance:

```
function getType<T>(myType : node) {
... // implementation for any unrecognized language
}
```

For those that know generic programming with C++ templates, here is a classical example of using template functions:

```
function f<1>() { return 1; }
function f<N>() { return $N*f<$N - 1$>()$; }
local f10 = f<10>();
if $f10 != 3628800$ error("10! should be worth 3628800");
traceLine("10! = " + f10);
```

Output:

10! = 3628800

We'll find below all useful functions we'll need for source code generation, including the template function getType<**T**>(myType : node) we spoke about:

```
   // file "GettingStarted/SharedFunctions.cws":
 1 function normalizeIdentifier(sName) {
 2    if sName {
 3       if startString(sName, "_")
 4          return "_" + normalizeIdentifier(subString(sName,
1));
 5       set sName = toUpperString(charAt(sName, 0))
 6                + subString(sName, 1);
 7       local iIndex = findFirstChar(sName, "_.");
 8       if !isNegative(iIndex) {
 9          local sNext = subString(sName, add(iIndex, 1));
10          return leftString(sName, iIndex)
11                + normalizeIdentifier(sNext);
12       }
13    }
14    return sName;
15 }
```

```
16
17 function getType<"C++">(myType :  node) {
18     local sType;
19     if myType.isObject set sType = myType.name + "*";
20     else if myType.name == "boolean" set sType = "bool";
21     else if myType.name == "string" set sType = "std::string";
22     else set sType = myType.name;
23     if myType.isArray set sType = "std::vector<" + sType + ">";
24     return sType;
25 }
26
27 function getParameterType<"C++">(myType :  node, sMode) {
28     local sType = getType<"C++">(myType);
29     if endString(sMode, "out") set sType += "&";
30     else if (sMode == "in") set sType = "const " + sType + "&";
31     return sType;
32 }
33
34 function getType<"JAVA">(myType :  node) {
35     local sType;
36     if myType.name == "string" set sType = "String";
37     else set sType = myType.name;
38     if myType.isArray set sType = "java.util.ArrayList/*<" +
sType + ">*/";
39     return sType;
40 }
41
42 function getParameterType<"JAVA">(myType :  node, sMode) {
43     return getType<"JAVA">(myType);
44 }
45
46 function getVariableName(sName, myType :  node) {
47     local sPrefix;
48     if myType.isArray set sPrefix = "t";
49     if myType.isObject set sPrefix += "p";
50     else {
51         switch(myType.name) {
52             case "int":  set sPrefix += "i";break;
53             case "double":  set sPrefix += "d";break;
54             case "boolean":  set sPrefix += "b";break;
55             case "string":  set sPrefix += "s";break;
56         }
57     }
58     return sPrefix + normalizeIdentifier(sName);
59 }
60
61 function getMethodID(myMethod :  node) {
62     local sMethodID = myMethod.name;
63     foreach i in myMethod.listOfParameters {
64         set sMethodID += "." + i.type.name;
65         if i.type.isArray set sMethodID += "[]";
```

```
66        }
67        return sMethodID;
68    }
```

LINE 1: this function normalizes identifiers, so as to capitalize the first letter and to suppress '_' or dots after capitalizing the letter that follows: `average_speed` becomes `AverageSpeed`, for example. This function is applied on attribute names for instance.

LINE 3: if the identifier starts with an underscore, it is preserved,

LINE 7: points to the first character encountered among an underscore and a dot,

LINE 17: this function returns the C++ type of a *Simple-Modeling* type node:

- an object returns a pointer to it,

- type `boolean` is written `bool` in C++,

- type `string` is written `std::string` in the C++ standard library,

- an array is written as an instantiated class of `std::vector`,

LINE 27: this function returns the C++ type of a *Simple-Modeling* type node as expected when passed to a method as a parameter type (`sMode` is worth *"in"*, *"out"*, *"inout"* or empty string),

LINE 34: this function returns the JAVA type of a *Simple-Modeling* type node:

- an object returns its class name,

- type `boolean` is written identically in JAVA,

- type `string` is written `String` in JAVA,

- an array is written as a `java.util.ArrayList` interface in JAVA,

LINE 42: this function returns the JAVA type of a *Simple-Modeling* type node as expected when passed to a method as a parameter type (`sMode` is worth *"in"*, *"out"*, *"inout"* or empty string, but we don't care about *"inout"* or *"out"* for the moment),

LINE 46: this function returns a variable name whose nomenclature depends on its type,

LINE 51: the `switch` statement allows selection among multiple sections of code, depending on the value of expression `myType.name`, enclosed in parentheses. If no controlling expression (announced by label `case`) matches with the value, and no `default` label is present, CODEWORKER throws an error.

LINE 61: this function returns a unique method ID, which is composed from the name of the method and the type of parameters, to avoid confusing protected areas from a method to another,

The next two examples both implement same functionalities, but in different languages (C++ and JAVA). They describe the skeleton of our objects.

3.4.1 C++ classes

A *pattern* script may be launched thanks to the procedure `generate` that expects three parameters:

- the first one is the file name of the script,

- the second one is the current context of execution that will be accessed via the `this` keyword into the script,

- the last one is the name of the file to generate,

The next *pattern* script describes the pattern of a C++ header file:

```
     // file "GettingStarted/CppObjectHeader.cwt":
1 #ifndef _@this.name@_h_
2 #define _@this.name@_h_
3
4 @
5 newFloatingLocation("include files");
6 @
7 // this line separates the two insertion points, so as to
distinguish them!
8 @
9 newFloatingLocation("class declarations");
10
11 function populateHeaderDeclarations(myType :  node) {
12     if myType.isObject insertTextOnce(getFloatingLocation("class
declarations"), "class " + myType.name + ";" + endl());
13     if myType.isArray insertTextOnce(getFloatingLocation("include
files"), "#include <vector>" + endl());
14     if myType.name insertTextOnce(getFloatingLocation("include
files"), "#include <string>" + endl());
15 }
16
17 @
18 class @this.name@ @
19 if existVariable(this.parent) {
20     insertTextOnce(getFloatingLocation("include files"),
"#include \"" + this.parent.name +".h\"" + endl());
21     @:  public @this.parent.name@ @
22 }
23 @ {
24     private:
25 @
26 foreach i in this.listOfAttributes {
27     populateHeaderDeclarations(i.type);
28     @ @getType<"C++">(i.type)@ _@getVariableName(i.name,
i.type)@;
29 @
30 }
31 @
32     public:
33         @this.name@();
34         ˜@this.name@();
35
36         // accessors:
37 @
38 foreach i in this.listOfAttributes {
39     local sVariableName = getVariableName(i.name, i.type);
40     %> inline <%getType<"C++">(i.type)%> get<%normalizeIdentifier
(i.name)%>() const { return _<%sVariableName%>; }
```

3.4. Generating code   33

```
41        inline void set<%normalizeIdentifier(i.name)@(<%getType
<"C++">(i.type)%> <%sVariableName@) { _<%sVariableName%> =
<%sVariableName%>; }
42  @
43  }
44  @
45        // methods:
46  @
47  foreach i in this.listOfMethods {
48      @ virtual @
49      if existVariable(i.type) {
50          populateHeaderDeclarations(i.type);
51          @@getType<"C++">(i.type)@@
52      } else {
53          @void@
54      }
55      @ @i.name@(@
56      foreach j in i.listOfParameters {
57          if !first(j) {
58              @, @
59          }
60          populateHeaderDeclarations(j.type);
61          @@getParameterType<"C++">(j.type, j.mode)@
@getVariableName(j.name, j.type)@@
62      }
63      @);
64  @
65  }
66  @
67      private:
68          @this.name@(const @this.name@&);
69          @this.name@& operator =(const @this.name@&);
70  };
71
72  #endif
```

LINE 1: the value of attribute $this$.name is written to the output file, where $this$ points to a node that describes the current class. Note that $this$ is facultative, and is assigned by the caller of procedure `generate` that runs this script.

LINE 5: put one anchor for including all files that we'll encounter as compulsory, while iterating attributes or methods. Example: if an attribute is an array, we'll need to include the STL header *vector* at this position of the file: *#include <vector>*. This insertion point is called `"include files"`.

LINE 6: to avoid that the two floating locations `"include files"` and `"class declarations"` (described just below) point to the same file position, an empty line is added,

LINE 9: put one anchor for announcing all classes that we'll encounter as referenced, while iterating attributes or methods. Example: if an attribute is an object *Planet*, we'll need to write *class Planet;* at this position of the file. This insertion point is called `"class declarations"`.

LINE 11: this function is called on every type encountered while iterating attributes and methods. Its role is to populate the `"include files"` and `"class declarations"` areas.

LINE 12: the type of an object must be declared at the beginning of the header, otherwise the compiler will not recognize it : the class is declared **once only** in the insertion point called `"class declarations"`. Use of function `insertTextOnce` assures that if this class has already been

inserted before, it will not be twice.

LINE 13: this type is an array, so the declaration of `std::vector` must be included to the insertion point called `"include files"`,

LINE 14: this type is a string, so the declaration of `std::string` must be included to the insertion point called `"include files"`,

LINE 19: if the class inherits from a parent class, this relationship must be written,

LINE 20: the parent class must be declared,

LINE 26: declaration of all attributes,

LINE 27: does the type of the attribute need some backward declarations?

LINE 38: accessors to each attribute,

LINE 40: there are two symbols to swap between writing a sequence of characters and interpreting script ; we have used the symbol **'@'**, and now we illustrate the use of tags **'<%** and **'%>**,

LINE 41: you can melt the two swapping symbol, but it is more difficult to read, so not very interesting!

LINE 47: declaration of all methods,

LINE 48: each method might be overloaded by subclasses,

LINE 49: the return type of the method is translated to C++,

LINE 50: does the return type of the method need some backward declarations?

LINE 51: expression `getType<"C++">(i.type)` to evaluate is embedded between double **'@'**. The first one allow swapping to the *sequence of characters* mode, but there is no characters to write. The second one allows swapping to the *script* mode, which is reduced just to evaluate the expression. The two final **'@'** take the same role as seen before.

LINE 56: parameters of the method are iterated to be written in C++

LINE 57: if iterator `j` doesn't point to the first parameter, a comma makes a separation with the precedent,

LINE 60: does the type of the parameter need some backward declarations?

Let's continue with the pattern that describes the skeleton of a C++ body file:

```
    // file "GettingStarted/CppObjectBody.cwt":
1   #ifdef WIN32
2   #pragma warning(disable :  4786)
3   #endif
4
5   @
6   setProtectedArea("include files");
7   @
8   #include "@this.name@.h"
9
10  @this.name@::@this.name@()@
11  local bAtLeastOne = false;
12  foreach i in this.listOfAttributes {
13      if !i.type.isArray && (i.type.name != "string") {
14          if bAtLeastOne {
15              @, @
16          } else {
17              @ :   @
18              set bAtLeastOne = true;
19          }
20          @_@getVariableName(i.name, i.type)@(@
21          if i.type.isObject {
22              @0L@
```

```
23              } else {
24                  switch(i.type.name) {
25                      case "int":
26                          @0@
27                          break;
28                      case "double":
29                          @0.0@
30                          break;
31                      case "boolean":
32                          @false@
33                          break;
34                  }
35              }
36          @)@
37      }
38  }
39  @  {
40  }

41
42  @this.name@::~@this.name@() {
43  @
44  foreach i in this.listOfAttributes {
45      if i.type.isAggregation && i.type.isObject {
46          local sAttributeName = "_" + getVariableName(i.name,
i.type);
47          local sIndex = "iterate" + normalizeIdentifier(i.name);
48          if i.type.isArray {
49              @ for (std::vector<@i.name@*>::const_iterator
@sIndex@ = @sAttributeName@.begin(); @sIndex@ !=
@sAttributeName@.end(); ++@sIndex@) {
50              delete *@sIndex@;
51          }
52  @
53          } else {
54              @ delete @sAttributeName@;
55  @
56          }
57      }
58  }
59  @}

60
61  @
62  foreach i in this.listOfMethods {
63      if existVariable(i.type) {
64          @@getType<"C++">(i.type)@@
65      } else {
66          @void@
67      }
68      @ @this.name@::@i.name@(@
69      foreach j in i.listOfParameters {
70          if !first(j) {
```

```
71          @, @
72        }
73        @@getParameterType<"C++">(j.type, j.mode)@
@getVariableName(j.name, j.type)@@
74        }
75      @) {
76 @
77          setProtectedArea(getMethodID(i));
78 @}
79 @
80  }
```

LINE 1: Visual C++-specific pragma must be added to prevent from intempestive warnings about template class instantiation of `std::vector<T>` in DEBUG mode!

LINE 6: the developer will add here all include files he will need for implementation of methods,

LINE 8: the header of this body is compulsory,

LINE 11: this part concerns the initialization of attributes. Some attributes, such as strings and vectors of the STL don't require to be initialized explicitly. It justifies the declaration of variable `bAtLeastOne` that is worth `false` as long as no attribute has been initialized yet. We'll see why below.

LINE 13: arrays and strings are skipped,

LINE 15: if it isn't the first attribute to be initialized, a comma make a separation with the precedent,

LINE 17: if it is the first attribute to be initialized, a colon is expected to announce the beginning of initializations

LINE 18: now, there is at least one attribute to be initialized,

LINE 21: attribute is populated with the default value corresponding to its type,

LINE 44: aggregated objects must be deleted before leaving this instance,

LINE 49: all elements of an aggregated array must be deleted

LINE 54: the aggregated object is deleted

LINE 62: implementation of all methods,

LINE 63: the return type of the method is translated to C++,

LINE 69: parameters of the method are iterated to be written in C++

LINE 70: if iterator `j` doesn't point to the first parameter, a comma makes a separation with the precedent,

LINE 77: a protected area is inserted, whose key is the method ID,

The leader script has to be improved to reclaim generation of C++ files:

CODEWORKER *command line to execute:*
```
-I Scripts/Tutorial -path .  -define DESIGN_FILE=GettingStarted/SolarSystem0.sml
-script GettingStarted/LeaderScript1.cws
```

```
    // file "GettingStarted/LeaderScript1.cws":
1 if !getProperty("DESIGN_FILE")
2    error("'-define DESIGN_FILE=file' expected on the command
line");
3 traceLine("'Simple Modeling' design file to parse = \""
4         + getProperty("DESIGN_FILE") + "\"");
5 parseAsBNF("GettingStarted/SimpleML-parsing.cwp",
6          project, getProperty("DESIGN_FILE"));
7 #include "TreeDecoration.cws"
8
9 #include "SharedFunctions.cws"
10 foreach myClass in project.listOfClasses {
```

```
11    traceLine("generating class '" + myClass.name + "' ...");
12    generate("GettingStarted/CppObjectHeader.cwt", myClass,
13        getWorkingPath() + "Scripts/Tutorial/GettingStarted/Cpp/"
14        + myClass.name + ".h");
15    generate("GettingStarted/CppObjectBody.cwt", myClass,
16        getWorkingPath() + "Scripts/Tutorial/GettingStarted/Cpp/"
17        + myClass.name + ".cpp");
18 }
```

LINE 9: all useful functions for source code generation are loaded here,

LINE 10: all classes are iterated and their C++ header and body are generated

LINE 12: instruction `generate` is applied on a *pattern* script and its second argument expects a node that will be seen as variable 'this' into the pattern script,

LINE 13: `getWorkingPath()` is worth the output path passed to the command line via the option '-path',

Output:

```
'Simple Modeling' design file to parse = "GettingStarted/SolarSystem0.sml"
file parsed successfully
generating class 'Planet' ...
generating class 'Earth' ...
generating class 'SolarSystem' ...
```

Let have a look on some generated files:

```
 // file "GettingStarted/Cpp/SolarSystem.h":
#ifndef _SolarSystem_h_
#define _SolarSystem_h_


#include <vector>
#include <string>


// this line separates the two insertion points, so as to
distinguish them!
class Planet;


class SolarSystem {
    private:
        std::vector<Planet*> _tpPlanets;


    public:
        SolarSystem();
        ~SolarSystem();


        // accessors:
        inline std::vector<Planet*> getPlanets() const { return
_tpPlanets; }
        inline void setPlanets(std::vector<Planet*> tpPlanets) {
_tpPlanets = tpPlanets; }


        // methods:
```

```
     private:
         SolarSystem(const SolarSystem&);
         SolarSystem& operator =(const SolarSystem&);
     };

     #endif

      // file "GettingStarted/Cpp/Planet.cpp":
   1 #ifdef WIN32
   2 #pragma warning(disable :  4786)
   3 #endif
   4
   5 //##protect##"include files"
   6 //##protect##"include files"
   7
   8 #include "Planet.h"
   9
  10 Planet::Planet() :  _dDiameter(0.0) {
  11 }
  12
  13 Planet::~Planet() {
  14 }
  15
  16 double Planet::getDistanceToSun(int iDay, int iMonth, int
iYear) {
  17 //##protect##"getDistanceToSun.int.int.int"
  18 //##protect##"getDistanceToSun.int.int.int"
  19 }
```

LINE 1: Visual C++-specific pragma must be added to prevent from intempestive warnings about template class instantiation of $std::vector<T>$ in DEBUG mode!

3.4.2 JAVA classes

Some modelers don't separate clearly the design and its implementation, but theoretically, no language-dependent data has to be included into the design. The modeling language should be improved to take into account some finer modeling aspects that lead to choose a mapping (for parameter types, for example) to the implementation language. The logic of a source code generation process is to factorize as most as possible the knowledge at the design level. We'll speak longer about it further.

Our design is totally independent from the implementation : a string isn't explicitly a const $std::string\&$ or a $std::string$ in C++, but the *pattern script* decides according to the context whether it is more judicious to choose the first C++ mapping or the second one.

This independence allows us implementing the same functionalities as in C++, but in JAVA now:

```
      // file "GettingStarted/JAVAObject.cwt":
   1 package solarsystem;
   2
   3 public class @this.name@ @
   4 if existVariable(this.parent) {
   5     @extends @this.parent.name@ @
   6 }
```

3.4. Generating code   39

```
 7  @{
 8  @
 9  foreach i in this.listOfAttributes {
10      @ private @getType<"JAVA">(i.type)@ _@getVariableName(i.name,
i.type)@;
11  @
12  }
13  @
14      public @this.name@() {}
15
16      // accessors:
17  @
18  foreach i in this.listOfAttributes {
19      local sVariableName = getVariableName(i.name, i.type);
20      @ public @getType<"JAVA">(i.type)@ get@normalizeIdentifier(i.name)@()
{ return _@sVariableName@; }
21      public void set@normalizeIdentifier(i.name)@(@getType<"JAVA">(i.type)@
@sVariableName@) { _@sVariableName@ = @sVariableName@; }
22  @
23  }
24  @
25          // methods:
26  @
27  foreach i in this.listOfMethods {
28      @ public @
29      if existVariable(i.type) {
30          @@getType<"JAVA">(i.type)@@
31      } else {
32          @void@
33      }
34      @ @i.name@(@
35      foreach j in i.listOfParameters {
36          if !first(j) {
37              @, @
38          }
39          @@getParameterType<"JAVA">(j.type, j.mode)@
@getVariableName(j.name, j.type)@@
40      }
41      @) {
42  @
43      setProtectedArea(getMethodID(i));
44  @ }
45
46  @
47  }
48  @}
```

LINE 4: if the class inherits from a parent class, this relationship must be written,
LINE 9: declaration of all attributes,
LINE 18: accessors to each attribute,
LINE 27: declaration of all methods,

The leader script has to be improved to reclaim generation of JAVA files:

CODEWORKER *command line to execute:*

```
-I Scripts/Tutorial -path .  -define DESIGN_FILE=GettingStarted/SolarSystem0.sml
-script GettingStarted/LeaderScript2.cws
```

```
      // file "GettingStarted/LeaderScript2.cws":
   1 if !getProperty("DESIGN_FILE")
   2     error("'-define DESIGN_FILE=file' expected on the command
line");
   3 traceLine("'Simple Modeling' design file to parse = \""
   4          + getProperty("DESIGN_FILE") + "\"");
   5 parseAsBNF("GettingStarted/SimpleML-parsing.cwp",
   6            project, getProperty("DESIGN_FILE"));
   7 #include "TreeDecoration.cws"
   8
   9 #include "SharedFunctions.cws"
  10 foreach myClass in project.listOfClasses {
  11     traceLine("generating class '" + myClass.name + "' ...");
  12     generate("GettingStarted/CppObjectHeader.cwt", myClass,
getWorkingPath() + "Scripts/Tutorial/GettingStarted/Cpp/" +
myClass.name + ".h");
  13     generate("GettingStarted/CppObjectBody.cwt", myClass,
getWorkingPath() + "Scripts/Tutorial/GettingStarted/Cpp/" +
myClass.name + ".cpp");
  14     generate("GettingStarted/JAVAObject.cwt", myClass,
getWorkingPath() + "Scripts/Tutorial/GettingStarted/JAVA/solarsystem/"
+ myClass.name + ".java");
  15 }
```

LINE 14: generates the JAVA implementation of the current design class,

Output:

```
'Simple Modeling' design file to parse = "GettingStarted/SolarSystem0.sml"
file parsed successfully
generating class 'Planet' ...
generating class 'Earth' ...
generating class 'SolarSystem' ...
```

Let have a look on some generated files:

```
      // file "GettingStarted/JAVA/solarsystem/SolarSystem.java":
   package solarsystem;

   public class SolarSystem {
       private java.util.ArrayList/*<Planet>*/ _tpPlanets;

       public SolarSystem() {}

       // accessors:
       public java.util.ArrayList/*<Planet>*/ getPlanets() { return
_tpPlanets; }
       public void setPlanets(java.util.ArrayList/*<Planet>*/
```

```
tpPlanets) { _tpPlanets = tpPlanets; }

          // methods:
    }
     // file "GettingStarted/JAVA/solarsystem/Planet.java":
    package solarsystem;

    public class Planet {
        private double _dDiameter;

        public Planet() {}

        // accessors:
        public double getDiameter() { return _dDiameter; }
        public void setDiameter(double dDiameter) { _dDiameter =
dDiameter; }

          // methods:
        public double getDistanceToSun(int iDay, int iMonth, int
iYear) {
    //##protect##"getDistanceToSun.int.int.int"
    //##protect##"getDistanceToSun.int.int.int"
        }

    }
```

3.5 Expanding a file

Expanding a file consists of generating code to some determined points of the file. These points are called `markups` and are noted **##markup##"***name-of-the-markup***"**, surrounded by comment delimiters.

For example, a valid markup inlayed in a C++ file could be:
```
//##markup##"factory"
```
and a valid markup inlayed in an HTML file could be:
```
<!- -##markup##"classes"- ->
```

Some data may accompany the markup. The block of data is put between tags `##data##`:
```
//##markup##"switch(sText)"
//##data##
//Customer
//Videostore
//##data##
```
You obtain the data attached to the current markup key by calling the function `getMarkupValue()` (see 4.6.14). This example extends the C++/Java functionalities with a *switch* statement working on a string expression.

A *pattern* script intended to expand code is launched thanks to the procedure `expand` that expects three parameters:

- the first one is the file name of the script,

- the second one is the current context of execution that will be accessed via the `this` keyword into

the script,

- the last one is the name of the file to expand,

Each time CODEWORKER will encounter a markup, it will call the *pattern* script that will decide how to populate it. The code generated by the *pattern* script for this markup is surrounded by tags **##begin##**"*name-of-the-markup*" and **##end##**"*name-of-the-markup*", automatically added by the interpreter. If some protected areas were put into the generated code, they are preserved the next time the expansion is required.

Note that CODEWORKER doesn't change what is written outside the markups and their begin/end delimiters.

Starting from a (very simple) HTML canvas, we'll generate an HTML documentation to our project *SolarSystem*. Here is the canvas that we would like to keep for all our projects:

```
   // file "GettingStarted/defaultDocumentation.html":
<HTML>
    <HEAD>
        <TITLE>some title...</TITLE>
    </HEAD>
    <BODY>
        <H1>some title...</H1>
        some global documentation...
<!-##markup##"classes presentation"->
    </BODY>
</HTML>
```

We'll copy it to *"GettingStarted/SolarSystem0.html"* to populate it with the characteristics of our current project. The *pattern* script that will be launched to expand *"GettingStarted/SolarSystem0.html"* is:

```
    // file "GettingStarted/HTMLDocumentation.cwt":
1  @
2  if getMarkupKey() == "classes presentation" {
3      foreach i in project.listOfClasses {
4          @
5          <H2><A href="#@i.name@">@i.name@</A></H2>
6  @
7          setProtectedArea(i.name + ":presentation");
8          if !isEmpty(i.listOfAttributes) {
9              @
10         <TABLE border="1" cellpadding="3" cellspacing="0"
width="100%">
11             <TR BGCOLOR="#CCCCFF">
12                 <TD><B>Type</B></TD>
13                 <TD><B>Attribute name</B></TD>
14                 <TD><B>Description</B></TD>
15             </TR>
16 @
17             foreach j in i.listOfAttributes {
18                 @ <TR>
19                 <TD>@composeHTMLLikeString(getType<this.language>
(j.type))@</TD>
20                 <TD>@j.name@</TD>
21                 <TD>
```

```
22   @
23             setProtectedArea(i.name + "::" + j.name +
":description");
24             @
25             </TD>
26         </TR>
27   @
28         }
29         @ </TABLE>
30   @
31       }
32       if !isEmpty(i.listOfMethods) {
33         @
34       <UL>
35   @
36         foreach j in i.listOfMethods {
37           @ <LI>@
38           if existVariable(j.type) {
39             @function @composeHTMLLikeString(getType
<this.language>(j.type))@ @
40           } else {
41             @procedure@
42           }
43           @<B>@j.name@</B>(@
44           foreach k in j.listOfParameters {
45             if !first(k) {
46               @, @
47             }
48             @@composeHTMLLikeString(getParameterType
<this.language>(k.type, k.mode))@ <I>@getVariableName(k.name,
k.type)@</I>@
49           }
50           @)
51           <BR>
52   @
53           setProtectedArea(i.name + "::" + getMethodID(j) +
":description");
54           @
55       </LI>
56   @
57         }
58       @ </UL>
59   @
60       }
61     }
62   }
```

LINE 2: the predefined function `getMarkupKey()` returns the name of the markup to expand,
LINE 3: the markup is worth "classes presentation", and so, we'll describe all classes
LINE 7: a protected area is embedded here, which has to be populated by hand into the expanded file for describing the class,
LINE 9: attributes are presented into a table,

LINE 18: the language into which types have to be expressed is given by `this.language`, and is worth "C++" or "JAVA" ; don't forget to convert the type to the HTML syntax, because of **'<'** or **'>'** to convert respectively to **'&lt'** or **'&gt'** for instance. Use the predefined function `composeHTMLLikeString()` to do this process.

LINE 23: a protected area is embedded here, which has to be populated by hand into the expanded file for describing the attribute,

LINE 37: methods are presented into unordered lists,

LINE 53: a protected area is embedded here, which has to be populated by hand into the expanded file for describing the method,

Now, we have to change the leader script, so as to take into account the generation of the documentation:

CODEWORKER *command line to execute:*
```
-I Scripts/Tutorial -path .  -define DESIGN_FILE=GettingStarted/SolarSystem0.sml
-script GettingStarted/LeaderScript3.cws
```

```
      // file "GettingStarted/LeaderScript3.cws":
 1  if !getProperty("DESIGN_FILE")
 2      error("'-define DESIGN_FILE=file' expected on the command
line");
 3  traceLine("'Simple Modeling' design file to parse = \""
 4          + getProperty("DESIGN_FILE") + "\"");
 5  parseAsBNF("GettingStarted/SimpleML-parsing.cwp",
 6          project, getProperty("DESIGN_FILE"));
 7  #include "TreeDecoration.cws"
 8
 9  #include "SharedFunctions.cws"
10  foreach myClass in project.listOfClasses {
11      traceLine("generating class '" + myClass.name + "' ...");
12      generate("GettingStarted/CppObjectHeader.cwt", myClass,
getWorkingPath() + "Scripts/Tutorial/GettingStarted/Cpp/" +
myClass.name + ".h");
13      generate("GettingStarted/CppObjectBody.cwt", myClass,
getWorkingPath() + "Scripts/Tutorial/GettingStarted/Cpp/" +
myClass.name + ".cpp");
14      generate("GettingStarted/JAVAObject.cwt", myClass,
getWorkingPath() + "Scripts/Tutorial/GettingStarted/JAVA/solarsystem/"
+ myClass.name + ".java");
15  }
16  if !existFile("Scripts/Tutorial/GettingStarted/SolarSystem0.html")
{
17      copyFile("Scripts/Tutorial/GettingStarted/defaultDocumentation.html",
"Scripts/Tutorial/GettingStarted/SolarSystem0.html");
18  }
19
20  local myDocumentationContext;
21  insert myDocumentationContext.language = "C++";
22  traceLine("generating the HTML documentation...");
23  setCommentBegin("<!-");
24  setCommentEnd("->");
25  expand("GettingStarted/HTMLDocumentation.cwt",
26          myDocumentationContext, getWorkingPath()
```

```
   27                 + "Scripts/Tutorial/GettingStarted/SolarSystem0.html");
```

LINE 16: copy the default empty HTML documentation to *"SolarSystem0.html"* if it doesn't exist yet,
LINE 20: the `myDocumentationContext` variable will be passed to the procedure `expand()`,
LINE 21: an attribute `language` is added to the `myDocumentationContext` variable, which specifies whether types must be expressed in C++ or in JAVA into the HTML documentation,
LINE 23: don't forget to specify comment delimiters that are expected by an HTML file,
LINE 25: the procedure `expand()` allow populating *"SolarSystem0.html"* with the characteristics of the project automatically,

Output:

```
'Simple Modeling' design file to parse = "GettingStarted/SolarSystem0.sml"
file parsed successfully
generating class 'Planet' ...
generating class 'Earth' ...
generating class 'SolarSystem' ...
generating the HTML documentation...
```

After executing this script, we obtain the following HTML documentation, where protected areas have to be populated, so as to describe classes and attributes and methods:

```
     // file "GettingStarted/SolarSystem0.html":
    <HTML>
        <HEAD>
            <TITLE>some title...</TITLE>
        </HEAD>
        <BODY>
            <H1>some title...</H1>
            some global documentation...
    <!-##markup##"classes presentation"-><!-##begin##"classes
presentation"->
            <H2><A href="#Planet">Planet</A></H2>
    <!-##protect##"Planet:presentation"-><!-##protect##"Planet:presentation"->
            <TABLE border="1" cellpadding="3" cellspacing="0"
width="100%">
                <TR BGCOLOR="#CCCCFF">
                    <TD><B>Type</B></TD>
                    <TD><B>Attribute name</B></TD>
                    <TD><B>Description</B></TD>
                </TR>
                <TR>
                    <TD>double</TD>
                    <TD>diameter</TD>
                    <TD>
    <!-##protect##"Planet::diameter:description"-><!-##protect##"Planet::diamet
                    </TD>
                </TR>
            </TABLE>

            <UL>
                <LI>function double <B>getDistanceToSun</B>(int
<I>iDay</I>, int <I>iMonth</I>, int <I>iYear</I>)
```

```
                    <BR>
        <!-##protect##"Planet::getDistanceToSun.int.int.int:description"-><!-##prot
                </LI>
            </UL>


            <H2><A href="#Earth">Earth</A></H2>
        <!-##protect##"Earth:presentation"-><!-##protect##"Earth:presentation"->
            <TABLE border="1" cellpadding="3" cellspacing="0"
width="100%">
                <TR BGCOLOR="#CCCCFF">
                    <TD><B>Type</B></TD>
                    <TD><B>Attribute name</B></TD>
                    <TD><B>Description</B></TD>
                </TR>
                <TR>
                    <TD>std::vector&lt;std::string&gt;</TD>
                    <TD>countryNames</TD>
                    <TD>
        <!-##protect##"Earth::countryNames:description"-><!-##protect##"Earth::cour
                    </TD>
                </TR>
            </TABLE>


            <H2><A href="#SolarSystem">SolarSystem</A></H2>
        <!-##protect##"SolarSystem:presentation"-><!-##protect##"SolarSystem:presen
            <TABLE border="1" cellpadding="3" cellspacing="0"
width="100%">
                <TR BGCOLOR="#CCCCFF">
                    <TD><B>Type</B></TD>
                    <TD><B>Attribute name</B></TD>
                    <TD><B>Description</B></TD>
                </TR>
                <TR>
                    <TD>std::vector&lt;Planet*&gt;</TD>
                    <TD>planets</TD>
                    <TD>
        <!-##protect##"SolarSystem::planets:description"-><!-##protect##"SolarSyste
                    </TD>
                </TR>
            </TABLE>
        <!-##end##"classes presentation"->
        </BODY>
    </HTML>
```

We'll suppose that the skeleton of the HTML documentation is acceptable for us. It will evolve with our design "SolarSystem0.sml": if some classes or some members are added or removed, the skeleton will take these changes into account. When the reference to a protected area disappears, because the member it was linked to changes its name or is removed, the protected area is kept up at the end of the file.

Now, we have to populate protected areas and parts of text that are put outside the markups, so as to complete our documentation. This work has been done to "SolarSystem1.html".

3.6 Translating a file

Up to now, we discovered parsing on one side and source code generation on the other side. The *translation* mode merges the two: it offers to parse a file conforming to a BNF and to translate it into another format, all in the same *translation* script.

A *translation* script looks like a *BNF-driven* parsing script, but where:

- special swapping character **'@'** or JSP-like tag **'<%'**,

- all functions and procedure intended to source code generation,

are allowed into compound statements that are announced by **'=>'**.

Outputs are written into another file, so the input file is preserved. The procedure that takes the translation in charge is called **translate()**.

Little practical example: all our documentation has been written in HTML, but we would like to translate it to LaTeX, into our own format. Why not?

First step, we must be able to read an HTML file according to a BNF representation. The corresponding BNF-driven script we have to write is restricted to be able to write our file "SolarSystem1.html":

```
     // file "GettingStarted/HTML-parsing.cwp":
 1  #noCase

 2

 3  HTML ::= #ignore(HTML) #continue '<' "HTML" '>' HTMLHeader
HTMLBody '<' '/' "HTML" '>' #empty;
 4  HTMLHeader ::= '<' #continue "HEAD" '>' [~['<' '/' "HEAD"
'>']]* '<' '/' "HEAD" '>';
 5  HTMLBody ::= '<' #continue "BODY" '>' HTMLText '<' '/' "BODY"
'>';
 6  HTMLText ::=
 7          [
 8              ~'<'
 9                  |
10              !['<' '/'] #continue '<'
11                  #readIdentifier:sTag HTMLNextOfTag<sTag>
12          ]*;
13  HTMLNextOfTag<"H1"> ::= #continue '>' HTMLText '<' '/' "H1"
'>';
14  HTMLNextOfTag<"H2"> ::= #continue '>' HTMLText '<' '/' "H2"
'>';
15  HTMLNextOfTag<"A"> ::= [HTMLAttribute]* #continue '>' HTMLText
'<' '/' 'A' '>';
16  HTMLNextOfTag<"TABLE"> ::= [HTMLAttribute]* #continue '>'
[HTMLTag("TR")]* '<' '/' "TABLE" '>';
17  HTMLTag(sTag :  value) ::= '<' #readText(sTag) #continue
HTMLNextOfTag<sTag>;
18  HTMLNextOfTag<"TR"> ::= [HTMLAttribute]* #continue '>'
[HTMLTag("TD")]* '<' '/' "TR" '>';
19  HTMLNextOfTag<"TD"> ::= [HTMLAttribute]* #continue '>' HTMLText
'<' '/' "TD" '>';
20  HTMLNextOfTag<"UL"> ::= [HTMLAttribute]* #continue '>'
[HTMLTag("LI")]* '<' '/' "UL" '>';
```

```
21  HTMLNextOfTag<"LI"> ::= [HTMLAttribute]* #continue '>' HTMLText
'<' '/' "LI" '>';
22  HTMLNextOfTag<"B"> ::= #continue '>' HTMLText '<' '/' "B" '>';
23  HTMLNextOfTag<"I"> ::= #continue '>' HTMLText '<' '/' "I" '>';
24  HTMLNextOfTag<"FONT"> ::= [HTMLAttribute]* #continue '>'
HTMLText '<' '/' "FONT" '>';
25  HTMLNextOfTag<"BR"> ::= ['/']?  #continue '>';
26  HTMLAttribute ::= #readIdentifier ['=' #continue
[STRING_LITERAL | WORD_LITERAL]]?;
27
28
29  STRING_LITERAL ::= #!ignore '\"' [~'\"']* '\"';
30  WORD_LITERAL ::= #!ignore [~['>' | '/' | ' ' | '\t']]+;
```

LINE 1: we don't care about the case: *<BODY>* and *<Body>* must be recognized as identical for instance,

LINE 6: the clause `HTMLText` reads the value between tags,

LINE 11: the best way to assure an easy extension of the grammar: to declare a template clause for describing the reading of a tag,

LINE 17: a clause to read a determined tag: the token `#readText` matches the input stream to the evaluated expression passed in parameter and the rest is read by the template clause that describes the reading of a tag,

Second step, we have to improve the BNF-driven script to add some features for generating the LaTeX code properly. Don't be afraid about the length of the source code, but go forward to the notes directly:

```
     // file "GettingStarted/HTML2LaTeX.cwp":
 1  #noCase

 2
 3  HTML2LaTeX ::= #ignore(HTML) #continue '<' "HTML" '>'
HTMLHeader HTMLBody '<' '/' "HTML" '>' #empty;
 4  HTMLHeader ::= '<' #continue "HEAD" '>' [~['<' '/' "HEAD"
'>']]* '<' '/' "HEAD" '>';
 5  HTMLBody ::= '<' #continue "BODY" '>' HTMLText '<' '/' "BODY"
'>';
 6  HTMLText ::= #!ignore
 7          [
 8              '&' #continue #readIdentifier:sEscape
HTMLEscape<sEscape> ';'
 9                  |
10              ~'<':cChar => writeText(cChar);
11                  |
12              !['<' blanks '/']
13              [
14                  "<!-" #continue [~"->"]* "->"
15                      |
16                  '<' #continue #ignore(HTML) #readIdentifier:sTag
HTMLNextOfTag<sTag>
17              ]
18          ]*;
19  HTMLEscape<"lt"> ::= => {@<@};
20  HTMLEscape<"gt"> ::= => {@>@};
```

```
21  HTMLTag(sTag :  value) ::= '<' #readText(sTag) #continue
HTMLNextOfTag<sTag>;
22  HTMLNextOfTag<"H1"> ::=
23       #continue '>' => {@\subsection{@}
24       HTMLText
25       '<' '/' "H1" '>' => {@}@};
26  HTMLNextOfTag<"H2"> ::=
27       #continue '>' => {@\subsubsection{@}
28       HTMLText
29       '<' '/' "H2" '>' => {@}@};
30  HTMLNextOfTag<"A"> ::= [HTMLAttribute]* #continue '>' HTMLText
'<' '/' 'A' '>';
31  HTMLNextOfTag<"TABLE"> ::=
32       [HTMLAttribute]* #continue '>' => {
33           @\begin{table@
34           newFloatingLocation("table PDF suffix");
35           @}{@
36           newFloatingLocation("table columns");
37           @}{.5}@
38       }
39       => local sPDFTableSuffix;
40       HTMLTableTitle(sPDFTableSuffix)
41       [HTMLTableLine(sPDFTableSuffix)]*
42       '<' '/' "TABLE" '>' => {@\end{table@sPDFTableSuffix@}
43  @};
44  HTMLTableTitle(sPDFTableSuffix :  node) ::=
45     '<' "TR" [HTMLAttribute]*
46     #continue '>'
47     [HTMLTableCol(sPDFTableSuffix)]*
48     '<' '/' "TR" '>' => {
49         insertText(getFloatingLocation("table PDF suffix"),
sPDFTableSuffix);
50         writeText(endl());
51     };
52  HTMLTableCol(sPDFTableSuffix :  node) ::=
53     '<' "TD" [HTMLAttribute]* #continue '>' => {
54         @{@
55         if !sPDFTableSuffix insertText(getFloatingLocation("table
columns"), "l");
56         else insertText(getFloatingLocation("table columns"),
"|l");
57         set sPDFTableSuffix += "i";
58     }
59     '<' 'B' '>' [#!ignore [~'<':cChar => writeText(cChar);]*]
'<' '/' 'B' '>'
60     '<' '/' "TD" '>' => {@}@};
61  HTMLTableLine(sPDFTableSuffix :  value) ::=
62       '<' "TR" [HTMLAttribute]* #continue '>' =>
{@\line@sPDFTableSuffix@@}
63         [HTMLTag("TD")]* '<' '/' "TR" '>' =>
{writeText(endl());};
```

```
64  HTMLNextOfTag<"TD"> ::=
65         [HTMLAttribute]* #continue '>' => {@{@}
66         HTMLCellText '<' '/' "TD" '>' => {@}@};
67  HTMLCellText ::= #!ignore
68         [
69             '&' #continue #readIdentifier:sEscape
HTMLEscape<sEscape> ';'
70                |
71             ['\r']?  ['\n'] => {@ @}
72                |
73             ~'<':cChar => writeText(cChar);
74                |
75             !['<' blanks '/']
76             [
77                 "<!-" #continue [~"->"]* "->"
78                    |
79                 '<' #continue #ignore(HTML) #readIdentifier:sTag
HTMLNextOfTag<sTag>
80             ]
81         ]*;
82  HTMLNextOfTag<"UL"> ::=
83         [HTMLAttribute]* #continue '>' => {@\begin{itemize}
84  @}
85         [HTMLTag("LI")]*
86         '<' '/' "UL" '>' => {@\end{itemize}
87  @};
88  HTMLNextOfTag<"LI"> ::=
89         [HTMLAttribute]* #continue '>' => {@\item @}
90         HTMLText
91         '<' '/' "LI" '>' => {writeText(endl());};
92  HTMLNextOfTag<"B"> ::=
93         #continue '>' => {@\textbf{@}
94         HTMLText
95         '<' '/' "B" '>' => {@}@};
96  HTMLNextOfTag<"I"> ::=
97         #continue '>' => {@\textbf{@}
98         HTMLText
99         '<' '/' "I" '>' => {@}@};
100 HTMLNextOfTag<"FONT"> ::= [HTMLAttribute]* #continue '>'
HTMLText '<' '/' "FONT" '>';
101 HTMLNextOfTag<"BR"> ::= ['/']?  #continue '>' => {
writeText(endl());};
102 HTMLAttribute ::= #readIdentifier ['=' #continue
[STRING_LITERAL | WORD_LITERAL]]?;
103
104
105 blanks ::= [' '| '\t' | '\r' | '\n']*;
106 STRING_LITERAL ::= #!ignore '\"' [~'\"']* '\"';
107 WORD_LITERAL ::= #!ignore [~['>' | '/' | ' ' | '\t']]+;
```

LINE 6: blank characters are interesting, so we refuse to ignore HTML blanks and comments,
LINE 8: handling of HTML escape sequences, announced by character **'&'**,

LINE 10: if not the beginning of a tag, the current character of the input stream is put to the output stream,

LINE 12: token operator '!' doesn't move the position of the input stream, and it continues in sequence only if the token expression that follows doesn't match; here, we check whether we have reached an end of tag or not,

LINE 14: we do not ignore comments anymore, so we have to do it my ourselves,

LINE 16: an embedded tag has been encountered,

LINE 19: template clauses `HTMLEscape<T>` are always valid and just convert special characters to their LaTeX representation,

LINE 22: in the real life, HTML tag *<H1>* could represent a chapter, but the LaTeX output file is intended to be included into the reference manual of CODEWORKER as an illustration ; it will be a part of a section, so chapters are translated as sub sections!

LINE 26: in the real life, HTML tag *<H2>* could represent a section, but for the same reason as above, it will be translated as a sub-sub section,

LINE 34: with HTML, the number of columns the table expects is deduced later. However, a latex table (well-formed for a PDF conversion) must know explicitly of how many columns it is composed. So, a floating position is attached to the current position of the output file. While discovering columns, text will be inserted here and further.

LINE 36: the format of each column is specified at this place,

LINE 39: we consider that the first line of the table gives the name of the columns, and we'll take the PDF table suffix ('ii' for 2 columns, 'iii' for 3 columns, ...) to write lines of the table correctly,

LINE 41: we translate as many lines of the table as we can read, knowing the PDF suffix,

LINE 52: the clause is intended to read the name of a column of a table, and to translate it to LaTeX, knowing that some text must be inserted into the declarative part of the LaTeX table,

LINE 67: the text into a cell of a table shouldn't contain paragraph jumps (empty line in LaTeX),

LINE 71: the simplest way to avoid empty lines is to ignore end of lines, and to replace it to a space,

Last step, we have to change the leader script, so as to take into account the translation of the HTML documentation to the LaTeX one:

CODEWORKER *command line to execute:*

```
-I Scripts/Tutorial -path . -define DESIGN_FILE=GettingStarted/SolarSystem0.sml
-script GettingStarted/LeaderScript4.cws
```

```
      // file "GettingStarted/LeaderScript4.cws":
 1  if !getProperty("DESIGN_FILE")
 2      error("'-define DESIGN_FILE=file' expected on the command
line");
 3  traceLine("'Simple Modeling' design file to parse = \""
 4          + getProperty("DESIGN_FILE") + "\"");
 5  parseAsBNF("GettingStarted/SimpleML-parsing.cwp",
 6              project, getProperty("DESIGN_FILE"));
 7  #include "TreeDecoration.cws"
 8
 9  #include "SharedFunctions.cws"
10  foreach myClass in project.listOfClasses {
11      traceLine("generating class '" + myClass.name + "' ...");
12      generate("GettingStarted/CppObjectHeader.cwt", myClass,
getWorkingPath() + "Scripts/Tutorial/GettingStarted/Cpp/" +
myClass.name + ".h");
13      generate("GettingStarted/CppObjectBody.cwt", myClass,
getWorkingPath() + "Scripts/Tutorial/GettingStarted/Cpp/" +
```

```
myClass.name + ".cpp");
   14      generate("GettingStarted/JAVAObject.cwt", myClass,
getWorkingPath() + "Scripts/Tutorial/GettingStarted/JAVA/solarsystem/"
+ myClass.name + ".java");
   15   }
   16
   17   local myDocumentationContext;
   18   insert myDocumentationContext.language = "C++";
   19   traceLine("generating the HTML documentation...");
   20   setCommentBegin("<!-");
   21   setCommentEnd("->");
   22   expand("GettingStarted/HTMLDocumentation.cwt",
   23         myDocumentationContext, getWorkingPath()
   24         + "Scripts/Tutorial/GettingStarted/SolarSystem1.html");
   25   translate("GettingStarted/HTML2LaTeX.cwp", project,
"GettingStarted/SolarSystem1.html", getWorkingPath() +
"Scripts/Tutorial/GettingStarted/SolarSystem.tex");
```

LINE 22: the procedure `expand()` will allow populating *"SolarSystem1.html"* with the characteristics
of the project,
LINE 25: a context of execution (`project` here) is given as a `this` variable, although no parsing will
be processed: reading and writing only, no data to keep,

Output:

```
'Simple Modeling' design file to parse = "GettingStarted/SolarSystem0.sml"
file parsed successfully
generating class 'Planet' ...
generating class 'Earth' ...
generating class 'SolarSystem' ...
generating the HTML documentation...
```

It generates the LaTeX file that composes the next sub section:

3.6.1   Design of a solar system

We dispose of some classes both in C++ and JAVA that allow building applications working on notions
of planets, stars and solar systems.

Planet

This class represents the characteristics of a planet.

Type	Attribute name	Description
double	diameter	the average diameter of the planet

- function double **getDistanceToSun**(int **iDay**, int **iMonth**, int **iYear**)

  This function returns the distance to the sun at a given trivial earthly date. This function reclaims
  more attributes for the planet, but we'll see it later (I'm afraid not!).

Earth

This class represents our planet, for instantiating our particular solar system for instance, and working on geopolitical data perhaps!

Type	Attribute name	Description
std::vector<std::string>	countryNames	the name of all countries are put into

SolarSystem

This class represents the solar system, with its constituents, the sun excluded for the moment.

Type	Attribute name	Description
std::vector<Planet*>	planets	the planets that compose the solar system.

3.7   The debugger

The -debug option passed to the command line allows running the interpreter in debug mode. See chapter 5.2.2 for more information about its functionalities. We'll apply it on our precedent *leader script*:

CODEWORKER *command line to execute:*

```
-I Scripts/Tutorial -path .  -define DESIGN_FILE=GettingStarted/SolarSystem0.sml
-script GettingStarted/LeaderScript5.cws -stdin GettingStarted/Debugger.cmd
-debug
```

```
    // file "GettingStarted/LeaderScript5.cws":
    if !getProperty("DESIGN_FILE")
        error("'-define DESIGN_FILE=file' expected on the command
line");
    traceLine("'Simple Modeling' design file to parse = \""
            + getProperty("DESIGN_FILE") + "\"");
    parseAsBNF("GettingStarted/SimpleML-parsing.cwp",
            project, getProperty("DESIGN_FILE"));
    #include "TreeDecoration.cws"

    #include "SharedFunctions.cws"
    foreach myClass in project.listOfClasses {
        traceLine("generating class '" + myClass.name + "' ...");
        generate("GettingStarted/CppObjectHeader.cwt", myClass,
getWorkingPath() + "Scripts/Tutorial/GettingStarted/Cpp/" +
myClass.name + ".h");
        generate("GettingStarted/CppObjectBody.cwt", myClass,
getWorkingPath() + "Scripts/Tutorial/GettingStarted/Cpp/" +
myClass.name + ".cpp");
        generate("GettingStarted/JAVAObject.cwt", myClass,
getWorkingPath() + "Scripts/Tutorial/GettingStarted/JAVA/solarsystem/"
+ myClass.name + ".java");
    }

    local myDocumentationContext;
```

```
        insert myDocumentationContext.language = "C++";
        traceLine("generating the HTML documentation...");
        setCommentBegin("<!-");
        setCommentEnd("->");
        expand("GettingStarted/HTMLDocumentation.cwt",
               myDocumentationContext, getWorkingPath()
               + "Scripts/Tutorial/GettingStarted/SolarSystem1.html");
        translate("GettingStarted/HTML2LaTeX.cwp", project,
"GettingStarted/SolarSystem1.html", getWorkingPath() +
"Scripts/Tutorial/GettingStarted/SolarSystem.tex");
```

Output:

```
"LeaderScript5.cws" at 5:  if !getProperty("DESIGN_FILE")
// The controlling sequence stops on the first statement of the
leader script.
// We go the next instruction:
n
"LeaderScript5.cws" at 7:  traceLine("'Simple Modeling' design file
to parse = \""
// twice more:
n2
'Simple Modeling' design file to parse = "GettingStarted/SolarSystem0.sml"
"LeaderScript5.cws" at 11:  parseAsBNF("GettingStarted/SimpleML-parsing.cwp",
//let plunge into the BNF-driven script:
s
parsed file is "e:/Projects/generator/Scripts/Tutorial/GettingStarted/SolarSyste
"SimpleML-parsing.cwp" at 6:  world ::= #ignore(C++)
[class_declaration]* #empty
//We are pointing to the beginning of the rule.  Let execute
'#ignore(C++)':
s
parsed file is "e:/Projects/generator/Scripts/Tutorial/GettingStarted/SolarSyste
"SimpleML-parsing.cwp" at 6:  world ::= #ignore(C++)
[class_declaration]* #empty
//Let go to the unbounded expression '[class_declaration]*':
s
parsed file is "e:/Projects/generator/Scripts/Tutorial/GettingStarted/SolarSyste
"SimpleML-parsing.cwp" at 6:  world ::= #ignore(C++)
[class_declaration]* #empty
//Now, we have a look to 'class_declaration':
s
parsed file is "e:/Projects/generator/Scripts/Tutorial/GettingStarted/SolarSyste
"SimpleML-parsing.cwp" at 16:  class_declaration ::= IDENT:"class"
#continue
//We visit 'INDENT:"class"' and we step over immediatly.  Into a
BNF-driven script, tokens of a
//sequence are iterated step by step, and 'next' runs all the
sequence in one shot:
s
parsed file is "e:/Projects/generator/Scripts/Tutorial/GettingStarted/SolarSyste
"SimpleML-parsing.cwp" at 112:  IDENT ::= #!ignore
```

```
['a'..'z'|'A'..'Z'|'_']
n
parsed file is "e:/Projects/generator/Scripts/Tutorial/GettingStarted/SolarSyste
"SimpleML-parsing.cwp" at 21:   IDENT:sClassName
//We visit 'INDENT:sClassName' and we step over immediatly:
s
parsed file is "e:/Projects/generator/Scripts/Tutorial/GettingStarted/SolarSyste
"SimpleML-parsing.cwp" at 112:   IDENT ::= #!ignore
['a'..'z'|'A'..'Z'|'_']
n
parsed file is "e:/Projects/generator/Scripts/Tutorial/GettingStarted/SolarSyste
"SimpleML-parsing.cwp" at 25:   => insert project.listOfClasses[sClassName].name
= sClassName;
//What about all local variables available on the stack?
l
sClassName
//What is the value of 'sClassName'?
t sClassName
Planet
//Now, we are looking at a classical statement of the language, an
'insert' assignment.  But
//it might be more convenient to see more source code:
d 4
parsed file is "e:/Projects/generator/Scripts/Tutorial/GettingStarted/SolarSyste
21:  IDENT:sClassName
22:  //note:  about parsing, classes are modeled into node
23:  //note:  \textbf{project.listOfClasses[}\textit{sClassName}\textbf{]}.
Its attribute
24:  //note:  \samp{name} contains the value of \textit{sClassName}.
25:  => insert project.listOfClasses[sClassName].name = sClassName;
26:  //note:  if the class inherits from a parent,
\samp{\textbf{':'}} is necessary followed by
27:  //note:  an identifier (pattern \samp{\#continue}), and the
identifier that matches with
28:  //note:  clause call \textit{IDENT} is assigned to the local
variable \samp{sClassName},
29:  [':'  #continue IDENT:sParentName
//What about the call stack?
stack
parsed file is "e:/Projects/generator/Scripts/Tutorial/GettingStarted/SolarSyste
"SimpleML-parsing.cwp" at 25:   => insert project.listOfClasses[sClassName].name
= sClassName;
parsed file is "e:/Projects/generator/Scripts/Tutorial/GettingStarted/SolarSyste
"SimpleML-parsing.cwp" at 6:   world ::= #ignore(C++)
[class_declaration]* #empty
parsed file is "e:/Projects/generator/Scripts/Tutorial/GettingStarted/SolarSyste
"LeaderScript5.cws" at 11:   parseAsBNF("GettingStarted/SimpleML-parsing.cwp",
//Exiting the debug session:
q
file parsed successfully
generating class 'Planet' ...
```

```
generating class 'Earth' ...
generating class 'SolarSystem' ...
generating the HTML documentation...
```

3.8   Scripts coverage and time consuming

The -quantify option passed to the command line allows running the interpreter with the profiling mode. See chapter 6.3 for more information about its functionalities. We'll apply it on our precedent *leader script*:

CODEWORKER *command line to execute:*
```
-I Scripts/Tutorial -path .  -define DESIGN_FILE=GettingStarted/SolarSystem0.sml
-script GettingStarted/LeaderScript6.cws -quantify
Scripts/Tutorial/GettingStarted/quantify.html
```

```
      // file "GettingStarted/LeaderScript6.cws":
    if !getProperty("DESIGN_FILE")
        error("'-define DESIGN_FILE=file' expected on the command
line");
    traceLine("'Simple Modeling' design file to parse = \""
            + getProperty("DESIGN_FILE") + "\"");
    parseAsBNF("GettingStarted/SimpleML-parsing.cwp",
             project, getProperty("DESIGN_FILE"));
    #include "TreeDecoration.cws"

    #include "SharedFunctions.cws"
    foreach myClass in project.listOfClasses {
        traceLine("generating class '" + myClass.name + "' ...");
        generate("GettingStarted/CppObjectHeader.cwt", myClass,
getWorkingPath() + "Scripts/Tutorial/GettingStarted/Cpp/" +
myClass.name + ".h");
        generate("GettingStarted/CppObjectBody.cwt", myClass,
getWorkingPath() + "Scripts/Tutorial/GettingStarted/Cpp/" +
myClass.name + ".cpp");
        generate("GettingStarted/JAVAObject.cwt", myClass,
getWorkingPath() + "Scripts/Tutorial/GettingStarted/JAVA/solarsystem/"
+ myClass.name + ".java");
    }

    local myDocumentationContext;
    insert myDocumentationContext.language = "C++";
    traceLine("generating the HTML documentation...");
    setCommentBegin("<!-");
    setCommentEnd("->");
    expand("GettingStarted/HTMLDocumentation.cwt",
           myDocumentationContext, getWorkingPath()
           + "Scripts/Tutorial/GettingStarted/SolarSystem1.html");
    translate("GettingStarted/HTML2LaTeX.cwp", project,
"GettingStarted/SolarSystem1.html", getWorkingPath() +
"Scripts/Tutorial/GettingStarted/SolarSystem.tex");
```

Output:

```
'Simple Modeling' design file to parse = "GettingStarted/SolarSystem0.sml"
file parsed successfully
generating class 'Planet' ...
generating class 'Earth' ...
generating class 'SolarSystem' ...
generating the HTML documentation...
```

Profiling results:

```
- quantify session -
quantify execution time = 179ms
User defined functions:
  populateHeaderDeclarations(...)  file "e:/Projects/generator/Scripts/Tutorial/(
at 29:  7 occurences in 0ms
  getMethodID(...)   file "e:/Projects/generator/Scripts/Tutorial/GettingStarted/:
at 98:  3 occurences in 0ms
  getParameterType(...)  file "e:/Projects/generator/Scripts/Tutorial/GettingSta
at 44:  3 occurences in 0ms
  getType(...)   file "e:/Projects/generator/Scripts/Tutorial/GettingStarted/Shar
at 31:  13 occurences in 0ms
  getVariableName(...)  file "e:/Projects/generator/Scripts/Tutorial/GettingStar
at 76:  26 occurences in 1ms
  normalizeIdentifier(...)  file "e:/Projects/generator/Scripts/Tutorial/Getting
at 5:  39 occurences in 0ms
Predefined functions:
  charAt(...):  39 occurrences
  composeHTMLLikeString(...):  7 occurrences
  endString(...):  9 occurrences
  endl(...):  19 occurrences
  executeStringQuiet(...):  1 occurrences
  existVariable(...):  11 occurrences
  findElement(...):  2 occurrences
  findFirstChar(...):  39 occurrences
  first(...):  12 occurrences
  getFloatingLocation(...):  23 occurrences
  getMarkupKey(...):  1 occurrences
  getProperty(...):  3 occurrences
  getWorkingPath(...):  11 occurrences
  isEmpty(...):  6 occurrences
  isNegative(...):  39 occurrences
  newFloatingLocation(...):  12 occurrences
  not(...):  72 occurrences
  startString(...):  39 occurrences
  subString(...):  39 occurrences
  toUpperString(...):  39 occurrences
Procedures:
  __RAW_TEXT_TO_WRITE(...):  498 occurrences
  clearVariable(...):  1 occurrences
  expand(...):  1 occurrences
  generate(...):  9 occurrences
  insertTextOnce(...):  24 occurrences
  parseAsBNF(...):  1 occurrences
```

```
  setCommentBegin(...):  1 occurrences
  setCommentEnd(...):  1 occurrences
  setProtectedArea(...):  19 occurrences
  traceLine(...):  5 occurrences
  translate(...):  1 occurrences
  writeText(...):  325 occurrences
Covered source code:  83%
- end of quantify session -
```

When the `-quantify` option isn't followed by an HTML file name, the synthetic profiling results are reported to the console:

- each user function appears, recalling the script file where it was defined, and giving how many times it was executed followed by the total execution time in milliseconds,

- each predefined function or procedure appears, giving how many times it was executed,

- the proportion of source code that was executed, considering visited scripts only,

If a file name was specified, the HTML output file highlights all visited script, so as to show parts of the code that are executed a lot and those that are less executed. Each visited line is prefixed by the number of times the controlling sequence has run on it.

Some points to notice:

- the function called `not` represents the unary boolean operator **!**,

- the instruction called `__RAW_TEXT_TO_WRITE` represents the text of a *pattern script* to put into the output stream directly, which is inlayed in **@...@** or in **%>...<%** tags,

- the instruction called `writeText` represents an expression of a *pattern script* that was inlayed in **@...@** or in **<%...%>** tags,

3.9   Translating interpreted scripts to C++ source code

Once the scripts are considered as stable, it might be interesting to convert the interpreter and all necessary scripts to an executable, for many reasons:

- the executable is largely faster than the interpreter, especially on big projects,

- a script file could be forgotten while delivering the project to somebody else,

- it is more convenient to handle an executable only rather than a set of script files and a long command line on the interpreter,

The executable is built starting from the corresponding C++ source codes of the script files. It exists two ways to ask for compiling the CODEWORKER script files to C++:

- the **−c++** option on the command line: it must be followed by the path of the directory where the C++ sources will be generated and the option `-script` or `-compile` must be set to specify the leader script to convert to C++,

- the **compileToCpp()** procedure (see 4.3.20): the required arguments are the leader script to convert to C++ and the directory where the C++ sources will be generated,

Compiling the project to C++ will convert the leader script and all its dependencies (meaning that all scripts that may be required by the leader will be compiled to C++) and then two makefiles will be created (a DSP for Visual C++ and a classical makefile intended to LINUX). The project takes the name of the leader script.

To compile our *Simple Modeling Language* project to C++, we may choose to proceed as one of the following:

- CODEWORKER command line to execute:
  ```
  -I Scripts/Tutorial -path .  -compile GettingStarted/LeaderScript6.cws
  -c++ Scripts/Tutorial/GettingStarted/bin
  ```

- instruction to execute in the console or into a script:

  ```
  compileToCpp("GettingStarted/LeaderScript6.cws",
  "Scripts/Tutorial/GettingStarted/bin", ".");
  ```

The directory called *"Scripts/Tutorial/GettingStarted/bin"* contains the C++ source files and the makefiles:

```
Scripts/Tutorial/GettingStarted/bin/CGExternalHandling.h
Scripts/Tutorial/GettingStarted/bin/CGRuntime.h
Scripts/Tutorial/GettingStarted/bin/CppObjectBody_cwt.cpp
Scripts/Tutorial/GettingStarted/bin/CppObjectBody_cwt.h
Scripts/Tutorial/GettingStarted/bin/CppObjectHeader_cwt.cpp
Scripts/Tutorial/GettingStarted/bin/CppObjectHeader_cwt.h
Scripts/Tutorial/GettingStarted/bin/CppParsingTree.h
Scripts/Tutorial/GettingStarted/bin/DynPackage.h
Scripts/Tutorial/GettingStarted/bin/HTML2LaTeX_cwp.cpp
Scripts/Tutorial/GettingStarted/bin/HTML2LaTeX_cwp.h
Scripts/Tutorial/GettingStarted/bin/HTMLDocumentation_cwt.cpp
Scripts/Tutorial/GettingStarted/bin/HTMLDocumentation_cwt.h
Scripts/Tutorial/GettingStarted/bin/JAVAObject_cwt.cpp
Scripts/Tutorial/GettingStarted/bin/JAVAObject_cwt.h
Scripts/Tutorial/GettingStarted/bin/LeaderScript6.dsp
Scripts/Tutorial/GettingStarted/bin/LeaderScript6_cws.cpp
Scripts/Tutorial/GettingStarted/bin/LeaderScript6_cws.h
Scripts/Tutorial/GettingStarted/bin/Makefile
Scripts/Tutorial/GettingStarted/bin/SimpleML-parsing_cwp.cpp
Scripts/Tutorial/GettingStarted/bin/SimpleML-parsing_cwp.h
Scripts/Tutorial/GettingStarted/bin/UtlException.h
```

The main C++ source file is *"LeaderScript6.cpp"* and the executable will be called *"LeaderScript6.exe"*.

The scripting language

CODEWORKER must be seen as a script interpreter that is intended to parse and to generate any kind of text or source code. This interpreter admits some options on the command line. Some of them look like those of a compiler.

CODEWORKER doesn't provide any Graphical User Interface, but a console mode allows interactivity with the user.

4.1   Command line of the interpreter

The *leader script* is the name given to the script that is executed first by the interpreter. It exists six ways to pass this leader script to the interpreter via the command line:

- the script describes all the processing tasks for parsing text, decorating the graph and generating code ; the option of the command line is `-script` to execute the script,

- the script describes an extended BNF grammar ; the option of the command line is `-parseBNF` for executing the script and parsing the source file,

- the script describes how to generate code ; the option of the command line is `-generate` to execute the script and to generate the output file,

- the script describes how to expand a file ; the option of the command line is `-expand` to execute the script and to expand the output file into its markups,

- a file contains embedded scripts driving their own expansion ; the option of the command line is `-autoexpand` to execute embedded scripts located below each markups, expanding the output file on markups,

- the script describes a *source-to-source* translation ; the option of the command line is `-translate` to execute the script and to translate the source file to the output file,

To find easier a file to open for reading among some directories, the option `-I` specifies a path to explore. It gives more flexibility in sharing input files (both scripts and user files, excepting generated or expanded files) between directories, and it avoids relative or absolute paths into scripts.

It is possible to define some properties on the command line, thanks to option `-define` (or `-D`). These properties are intended to be exploited into scripts.

It is recommended to specify a kind of *working directory* with option `-path`. The assigned value is accessible into scripts via the function `getWorkingPath()`. This *working directory* generally indicates the output path for copying or generating files. The developer of scripts decides how to use it.

CODEWORKER interprets scripts efficiently for speed. However, it is more convenient to run a standalone executable, instead of the interpreter and some script files. Moreover, once scripts are stable, why not to compile them as an executable to run the project a few times faster? Option -c++ allows translating the leader script and all its dependencies to *C++* source codes, ready-to-compile.

To facilitate the tracking of errors, an integrated debugger is called thanks to the option -debug. It runs into the console, and some classical commands allow taking the control of the execution and exploring the stack and the variables.

Here are presented all switches that are allowed on the command line:

Switch	Description
-args [*arg*]*	Pass some arguments to the command line. The list of arguments stops at the end of the command line or as soon as an option is encountered. The arguments are stored in a global array variable called _ARGS.
-autoexpand *file-to-expand*	The file *file-to-expand* is explored for expanding code at markups, executing a *template-based* script inserted just below each markup. It is identical to execute the script function **autoexpand(***file-to-expand***, project)**.
-c++ *generated-project-path* *CodeWorker-path*?	To translate the leader script and all its dependencies in C++ source code, once the execution of the leader script has achieved (same job as compileToCpp() 4.3.20). The *CodeWorker-path* is optional and gives the path through includes and libraries of the software. However, it is now recommended to specify *CodeWorker-path* by the switch -home.
-c++2target *script-file* *generated-project-path target-language*?	To translate the leader script and all its dependencies in C++ source code. Hence, the C++ is translated to a target language, all that once the execution of the leader script has achieved. Do not forget to give the path through includes and libraries of CodeWorker, setting the switch -home. A preprocessor definition called **"c++2target-path"** is automatically created. It contains the path of the generated project. Call getProperty("c++2target-path") to retrieve the path value. *target-language* is optional if at least one script of the project holds the target into its filename, just before the extension. Example: "myscript.java.cwt" means that the target language of this script is "java". A property can follow the name of the target language, separated by a '=' symbol. The property is accessible via getProperty("c++2target-property"), and its nature depends on the target. For instance, in Java, this property represents the package the generated classes will belong to. Example: java=org.landscape.mountains.
-c++external *filename*	To generate C++ source code for implementing all functions declared as external into scripts.
-commentBegin *format*	To specify the format of a beginning of comment.
-commentEnd *format*	To specify the format of a comment's end.
-compile *scriptFile*	To compile a script file, just to check whether the syntax is correct.

Switch	Description
`-commands` *commandFile*	To load all arguments processed ordinary on the command-line. It must be the only switch or else passed on the command-line.
`-console`	To open a console session (default mode if no script to interpret is specified via `-script` or `-compile` or `-generate` or `-expand`.
`-debug` [*remote*]?	To debug a script in a console while executing it. The optional argument *remote* defines parameters for a remote socket control of the debugging session. *remote* looks like **<hostname>:<port>**. If <hostname> is empty, CodeWorker runs as a socket server.
`-define VAR=`*value* or `-D ...`	To define some variables, as when using the C++ preprocessor or when passing properties to the JAVA compiler. These variables are similar to properties, insofar as they aren't exploited during the preprocessing of scripts to interpret. This option conforms to the format `-define VAR` when no value has to be assigned ; in that case, *"true"* is assigned by default to variable `VAR`. The script function **getProperty("VAR")** gives the value of variable `VAR`.
`-expand` *pattern-script* *file-to-expand*	Script file *pattern-script* is executed to expand file *file-to-expand* into markups. It is identical to execute script function **expand(***pattern-script***, project,** *file-to-expand***)**.
`-fast`	To optimize speed. While processing generation, the output file is built into memory, instead of into a temporary file.
`-generate` *pattern-script* *file-to-generate*	Script file *pattern-script* is executed to generate file *file-to-generate*. It is identical to execute script function **generate(***pattern-script***, project,** *file-to-generate***)**.
`-genheader` *text*	Adds a header at the beginning of all generated files, followed by a text (see procedure `setGenerationHeader()` 4.3.183).
`-help` or ?	Help about the command line.
`-home` *CodeWorker-path*	Specifies the path to the home directory of CodeWorker.
`-I` *path*	Specify a path to explore when trying to find a file while invoking `include` or `parseFree` or `parseAsBNF` or `generate` or `expand` or ... This option may be repeated to specify more than one *path*.
`-insert` *variable_expression* *value*	Creates a new node in the main parse tree `project` and assigns a constant value to it. It is identical to execute the statement **insert** *variable_expression* **= "** *value* **" ;**.
`-nologo`	The interpreter doesn't write the copyright in the shell at the beginning.

Switch	Description
`-nowarn` *warnings*	Specified warning types are ignored. They are separated by pipe symbols. Today, the only recognized type is **undeclvar**, which prevents the developer against the use of a undeclared variable.
`-parseBNF` *BNF-parsing-script source-file*	The script file *BNF-parsing-script* parses *source-file* from an extended BNF grammar. It is identical to execute the script function **parseAsBNF(***BNF-parsing-script***, project, *source-file***)**.
`-path` *path*	Output directory, returned by the script function **getWorking-Path()**, and used ordinary to specify where to generate or copy a file.
`-quantify` [*outputFile*]?	To execute scripts into quantify mode that consists of measuring the coverage and the time consuming. Results are saved to HTML file *outputFile* or displayed to the console if not present.
`-report` *report-file request-flag*	To generate a report once the execution has achieved. The report is saved to file *report-file* and nature of information depends on the flag *request-flag*. This flag must be built by computing a bitwise OR for one or several of the following integer constants: • **1**: provides every output file written by a template-based script (*generate()*, *expand()* or *translate*) • **2**: provides every input file scanned by a BNF parse script (*parseAsBNF()* or *translate()*) • **4**: provides details of coverage recording for every output file using the **#coverage** directive • **8**: provides details of coverage recording for every input file using the **#matching** directive • **16**: provides details of coverage recording for every output file written by a template-based script • **32**: provides details of coverage recording for every input file scanned by a BNF parse script Notice that flags *16* and *32* may become highly time and memory consuming, depending both on how many input/output files you have to process and on their size.
`-script` *script-file*	Defines the *leader* script, which will be executed first.
`-stack` *depth*	To limit the recursive call of functions, for avoiding an overflow stack memory. By default, the *depth* is set to 1000.
`-stdin` *filename*	To change the standard input for reading from an existing file. It may be useful for running a scenario.
`-stdout` *filename*	To change the standard output for writing it to a file.
`-time`	To display the execution time expressed in milliseconds, just before exiting.

Switch	Description
-translate *translation-script source-file file-to-generate*	Script file *translation-script* processes a *source-to-source* translation. It is identical to execute the script function **translate**(*translation-script*, **project,** *source-file*, *file-to-generate*).
-varexist	To trigger a warning when the value of a variable that doesn't exist is required into a script.
-verbose	To display internal messages of the interpreter (information).
-version *version-name*	To force interpreted scripts as written in a precedent version given by *version-name*.

Note that the interpreter proposes a convenient way for running a common script with arguments:
```
codeworker <script-file> <arg1> ...  <argN> [<switch>]*
```

This writing replaces the more verbose:
```
codeworker -script <script-file> -args <arg1> ...  <argN>
[<switch>]*
```

A console mode is launched when the command line is empty. The console only accepts scripts written in the common syntax, with common functions and procedures. So, parsing and generation scripts aren't typed directly on the console.

4.2   Syntax generalities and statements

A script in CODEWORKER consists of a series of statements that are organized into *blocks* (also known as *compound statements*). A statement is an instruction the interpreter has to execute.

A single statement must close with a semicolon (**';'**). A compound statement is defined by enclosing instructions between braces (**'{}'**). A *block* can be used everywhere you can use a single statement and must never end with a semicolon after the trailing brace.

Comments are indicated either by surrounding the text with **'/*'** and **'*/'** or by preceding the rest of the line to ignore with a double slash (**'//'**).

It exists three families of scripts here. To facilitate their syntax highlighting in editors, or to indicate briefly the type of the script, we suggest to employ some file extensions, depending on the nature of the script. The next table exposes the different extensions used commonly in CodeWorker.

Extension	Description
".cwt"	a *template-based* script, for text generation
".cwp"	a *extended-BNF* parse script, for parsing text
".cws"	a *common* script, none of the precedent

The structure of the grammar is so rich that it is a challenge to find an editor, which offers a syntax highlighting engine powerful enough. **JEdit** proposes the writing of production rules to describe it, so it is possible to express the syntax highlighting of the scripting language.

You'll find a package dedicated to **JEdit** on the Web site, for the inclusion of these new highlighting modes. Many thanks to Patrick Brannan for this contribution.

4.2.1   preprocessor directives

A preprocessor directive always starts with a **'#'** symbol and is followed by the name of the directive.

Including a file

The **#include** *filename* directive tells the preprocessor to replace the directive at the point where it appears by the contents of the file specified by the constant string *filename*. The preprocessor looks for the file in the current directory and then searches along the path specified by the -I option on the command line.

Extending the language via a package

A *package* is an extension of the scripting language that allows adding new functions in CODEWORKER at runtime. A package is implemented as an executable module, which exports all new functions the developer wants to make available in the interpreter.

Loading of a package

The preprocessor directive #use tells the interpreter that it must extend itself with the functions exposed by a package.

The syntax is: **#use** *package-name*

Loading a package more than once has no effect.

The name of the package must prefix the name of the function, when calling it: *package-name*::*my-function*(*parameters...*)

Example:
```
#use PGSQL
PGSQL::connect("-U pilot -d emergencyDB");
local sRequest = "SELECT solution FROM average_adjustment WHERE
damage = 'broken wing'";
local listOfSolutions;
PGSQL::selectList(sRequest, listOfSolutions);
if listOfSolutions.empty()
  traceLine("No solution.  Suggestion:  parachute jump?");
else {
  traceLine("Solutions:");
  foreach i in listOfSolutions
    traceLine(" -" + i);
}
PGSQL::disconnect(); // if the plane hasn't crashed yet
```

The PGSQL package serves here for connecting to and querying a PostGreSQL database. For this example, the package exports three functions: *PGSQL::connect*, *PGSQL::selectList* and *PGSQL::disconnect*.

The executable module

CODEWORKER expects a dynamic library, whose name is deduced from the package name and from the platform the interpreter is running to.
The short name of the dynamic library concatenates **"cw"** at the end of the package name. The extension of the dynamic library must be **".dll"** under *Microsoft Windows*, and **".so"** under *Linux*.

You must put the dynamic library at a place where CODEWORKER will find it at runtime.
Microsoft Windows proceeds in the following order to locate the library:

- The directory where the executable module for the current process is located.

Chapter 4. The scripting language

- The current directory.

- The Windows system directory (not recommended - it concerns CODEWORKER only).

- The Windows directory (not recommended - same reason).

- The directories listed in the PATH environment variable.

Under *Unix*, a relative path for the shared object refers to the current directory (according to the man description of `dlopen(3C)`).

So, when CODEWORKER reads **#use** PGSQL, it searches a dynamic library called `"PGSQL`**cw.dll**`"` under *Windows* or `"PGSQL`**cw.so**`"` under *Linux*.

Building a package

This section is intended to those that want to build their own packages, for binding to a database or to a graphical library ... or just for gluing with their own libraries.

When the interpreter find the preprocessor directive **#use** *package-name* in a script, it loads the executable module and executes the exported C-like function **CW4DL_EXPORT_SYMBOL void** *package-name*_**Init(CW4dl::Interpreter*)**.

The preprocessor definition `CW4DL_EXPORT_SYMBOL` and the namespace `CW4dl` are both declared in the C++ header file `"CW4dl.h"`. This header file is located in the `"include"` directory if you downloaded binaries, and at the root of the project if you downloaded sources.

The C-like function '*package-name*_**Init**`()`' MUST be present! *C-like* means that it is declared extern "C" (done by *CW4DL_EXPORT_SYMBOL*).

Initializing the module that way is useful for registering new functions in the engine, via the function `createCommand()` of the interpreter (see the header file `"CW4dl.h"` in the declaration of the class `Interpreter` for learning more about it).

Every function to export must start its declaration with the preprocessor definition `CW4DL_EXPORT_SYMBOL` (means 'extern "C"', but a little more under *Windows*).

- Up to 4 parameters, the signature of such a function looks like:
  CW4DL_EXPORT_SYMBOL const char*
  `selectList(`**CW4dl::Interpreter*,**
  　　　CW4dl::Parameter p1, **CW4dl::Parameter** p2);

  where *selectList* is a function expecting 2 parameters.

  The initializer `PGSQL_Init()` in our example informs the engine about the existence of this function *selectList* in the package:
  `createCommand("selectList", VALUE_PARAMETER, NODE_PARAMETER);`
  which means that *selectList* expects a string followed by a tree.

  In the body of the function 'selectList(...)', the C++ binding is obtained easily by a cast of `CW4dl::Parameter`:

  - `(const char*)` p1 for the value parameter *p1*,
  - `(CW4dl::Tree*)` p2 for the node parameter *p2*,

- if a function contains strictly more than 4 parameter, its signature changes and requires a variable number of parameters:　**CW4DL_EXPORT_SYMBOL const char***
  `myFunction(`**CW4dl::Interpreter*,**
  　　　int nbParams, **CW4dl::Parameter*** tParams);

---

where `tParams` is an array of parameter types, and where 'nbParams' gives the size.

The initializer `PGSQL_Init()` informs the engine about the existence of this function in the package differently too: `createCommand("myFunction", 6, tParams);`
which means that *myFunction* has 6 parameters whose types are provided in *tParams*.

Every function returns **const char***. The CodeWorker's keyword **null** designates an atypical tree node. It doesn't accept navigation and reference, only passing by parameter to a function. On the C++ side, this *null* tree node is seen as a null pointer of kind *CW4dl::Tree**.

The interpreter **CW4dl::Interpreter** represents the runtime context of CodeWorker. It is the unavoidable intermediary between the module you are building and CodeWorker.
Use it for:

- registering new functions into the CodeWorker's engine,

- throwing an error,

- handling parse trees,

The **#line** directive forces to another number the line counter of the script file being parsed. The line just after the directive is supposed to be worth the number specified after `#line`.

Changing the syntax of the scripting language

The **#syntax** directive tells the preprocessor not to parse the following instructions as classical statements of the scripting language, but as conforming to another syntax. It allows adapting the syntax to what you are programming:

- If you are programming a kind of *makefile* logic, where you have to check whether a file has been changed before another or not (using the function `fileLastModification()` 4.3.71 for example), it is clear that you would prefer to implement it in a *makefile-like* syntax rather than in the scripting language's syntax,

- If you are programming a kind of *shell* logic, where you have to copy files and directories, or re/move them, you would prefer to implement it in a *shell-like* syntax rather than in the scripting language's syntax. For instance:
  ```
  traceLine("Creating directory 'CodeWorker'...");
  removeDirectory("CodeWorker");
  copyFile("readme.txt", "CodeWorker/readme.txt");
  ...
  ```
  might be written in a *shell-like* syntax, inlayed in the CODEWORKER script:
  ```
  #syntax shell:"TinyShell.cwp"
  echo Creating directory 'CodeWorker'...
  rmdir CodeWorker
  copy readme.txt CodeWorker/readme.txt
  ...
  #end syntax
  ```

The directive admits the following writing:
"#syntax" [*parsing-mode* [`':'` *BNF-script-file*]? | *BNF-script-file*]

How does it work? The piece of source code, which doesn't conform to the syntax of the script language, is put between the directives `#syntax ...` and `#end syntax`. If the trailing directive isn't found,

the remaining of the script is considered as written in a foreign syntax. Be careful that the trailing directive must start at the beginning of the line necessary to be recognized and that no spaces are allowed between # and end.

At runtime, the famous piece of source code is parsed and processed via the BNF script file.

Note that it is possible to attach an identifier (called *parsing-mode* above) to a script file, and to specify later, in any other script, the *parsing mode* only; CODEWORKER will find the corresponding BNF script file. It avoids to handle a physical name of the BNF parsing file, where a logical name of parsing mode is more convenient.

Example:

```
    // the first time, a parsing mode may be attached to the BNF
script file
    #syntax shell:"TinyShell.cwp"
    ...
    #end syntax

    // at the second call, it isn't recommended to use the path of
the parsing file
    // it is better to use the parsing mode registered previously
    #syntax shell
    ...
    #end syntax

    // here, I know that I'll call it once only, so I don't care
about a parsing mode
    #syntax "MakeFile.cwp"
    ...
    #end syntax
```

where the parsing script *"TinyShell.cwp"* might be worth:

```
    // file "GettingStarted/TinyShell.cwp":
 tinyShell ::=
        #ignore(C++)
        #continue
        [
            #readIdentifier:sCommand
            #ignore(blanks) #continue
            command<sCommand>
        ]* #empty;

    //------------------//
    // commands of the tiny shell //
    //------------------//
    command<"copy"> ::=
        #continue parameter:sSource parameter:sDestination
        => {copyFile(sSource, sDestination);};

    command<"rmdir"> ::=
        #continue parameter:sDirectory
        => {removeDirectory(sDirectory);};
```

4.2. Syntax generalities and statements   69

```
command<"del"> ::=
        #continue parameter:sFile
        => {deleteFile(sFile);};


//------------
// Some useful clauses
//------------
parameter:value ::=
        #readCString:parameter
            |
        #!ignore #continue [~[' ' | '\t' | '\r' |
'\n']]+:parameter;
```

Of course, the parsing and the processing are implemented in the scripting language, so changing the syntax will be slower than keeping the default one. However, it allows writing a code easy to support and to understand.

Managing changes in a multi-language generation

The directives `#reference` and `#attach` serve to be notified when a change has been made into a script for generating in a given language, but not taken back in another language. For example, you are writing a framework both in C++ and JAVA. You are adding some new features in C++ or correcting some mistakes. One day, you'll be care not to forget to update the JAVA generation. In fact, thanks to these directives, a warning will be produced up to changes will have been put in the other script.

How does it work? Directives must delimit the piece of script you have changed:
"#reference" *key*

...

"#end" *key*

The key is an identifier that allows putting more than one *reference* area into a script file. A *#reference* area might cover one or more *#reference* directives, without confusing about boundaries. The directive must be put at the beginning of the line.

Here are the directives delimiting the piece of script that should be updated later in another file:
"#attach" *reference-file ':' reference-key*

...

"#end" *reference-key*

A *#attach* area might cover one or more *#reference* or *#attach* directives, as a *#reference* area. The directive must be put at the beginning of the line.

The first time CODEWORKER will encounter the *reference* script file, it will compute a number that depends on the content of the area. The first time CODEWORKER will encounter an *attached* script file, it will get back the *magic* number of the reference area, found both by the file name and the key of the reference. And then, at the beginning, the *reference* and *attached* areas are considered as similar. CODEWORKER stores the magic number of the reference just behind the `#attach` directive:
"#attach" *reference-file ':' reference-key ',' reference-number*

In fact, a script file that must be updated, so as to store the magic numbers for some attached areas, takes into account the modifications at the end of the parsing, and only if no error was encountered. If the `writefileHook()` function (see 4.2.6) is implemented, it is called and the script file doesn't change if it returns `false`. If the script file is read-only, the corresponding `readonlyHook()` function is called (see 4.2.6). If it isn't possible to save the script file, an error is thrown.

When a change occurs in the *reference* area, the next time CODEWORKER will encounter it, the magic number will be recomputed. When an attached piece of script is encountered after the change, the old magic number of the reference is compared to the new one. If they aren't the same, a warning is displayed to notify that the attached area hasn't been updated yet.

Once the changes have been taken back into the attached area, the magic number of the reference must be cut (don't forget the comma too!). And so, the next time this attached area will be encountered by the interpreter, it will get back the *magic* number of the reference area. And then, the *reference* area and the *attached* area are considered as similar once again.

Of course, the use of these directives is quite constraining. However, it is the only way in CODEWORKER to assure that features and corrections have been taken back in all generated languages.

4.2.2   Constant literals

CODEWORKER handles all basic types as strings, and doesn't distinguish a double from a boolean or a date. A `string` literal is a sequence of characters from the source character set enclosed in double quotation marks (`" "`). String literals are used to represent a sequence of characters which, taken together, form a null-terminated string. The interpretation done of the data depends on the context: function `increment(index)` expects that its argument `index` contains a number, but stored as a string.

- Floating-point numbers are represented as they are commonly admitted into programming languages: *3.141592* or *5.5E+6*.

- Integers are represented without the dot, *64* for instance.

- A character literal is represented between single quotes as in C or JAVA; it admits classical escape characters.

- Bytes are represented as a couple of hexadecimal digits. The *4D* byte is the ASCII of the letter *N*.

- About boolean types, an empty string `""` means *false*, and any kind of sequence of characters means *true*, such as "1" or "raspberries". Two constant literals are provided: keyword **true** is worth *"true"* and **false** is an empty string.

- dates are written according to a format that looks like *24sep2002*, where:

  - day takes 2 digits

  - month is represented as the 3 first letters of the corresponding english word ; *aug* as *august* and *may* as *may*, for instance

  - year takes 4 digits: *2002* but never *02*.

- the time representation conforms to the format:
  HH:MM:SS.millis

A *constant tree* describes a tree as a list of constant trees and expressions, intended to be assigned to a variable. Example:
```
local aVariable = "a"{["yellow", "red":"or".alternative="orange"],
.vehicle="submarine"};
```
You'll find more information in the sub section **??** below.

4.2.3 Variables, declaration and assignment

Variables serve as containers for the data you use into scripts. Data type is a tree that may be reduced to a leaf node, which contains a value and that's all.

Declaring variables

It isn't necessary to declare a variable before using if for the first time. A variable that is assigned without being declared is understood as a new sub-node to be added to the current tree context. The current context is obtained by the read-only variable called **this**. It corresponds to the main parse tree whose root name is **project** when you are into the leader script, and to the variable passed by parameter when calling a parsing or pattern script.

The next table exposes all pre-defined variable names (accessible from anywhere) and their meaning:

Variable Name	Description
project	The main parse tree, always present.
this	It points to the current context variable.
_ARGS	An array of all custom command-line arguments. Custom arguments are following the script file name or the switch `-args` on the command-line.
_REQUEST	If the interpreter works as a CGI program, it stores all parameters of the request in a association table. The key is the parameter name, which associates the corresponding value.

A variable that is read without being declared returns an empty string, but doesn't cause the creation of a sub-node. The danger is that you aren't safe from a spelling mistake. To prevent it, put the option `-varexist` on the command line and use the function `existVariable()` to check whether a variable exists or not.

Scope

When you declare a local variable, it is valid for use within a specific area of code, called the *scope*. When the flow of execution leaves the scope, the content of the variable, a subtree specially allocated during its declaration, is deleted and disappears forever from the stack. A scope is delimited by a *block*.

To declare a variable to the stack, use the following declaration statement:
```
local-variable-statement ::= "local" local-variable-declaration ';'
local-variable-declaration ::= variable [ '=' assignment-expression
]?
assignment-expression ::= constant-tree | expression
constant-tree ::= [tree-value]? '{' [tree-array-or-attribute [','
tree-array-or-attribute]* ]? '}'
tree-value ::= expression
tree-array-or-attribute ::= tree-array | tree-attribute
tree-attribute ::= '.' attribute-name '=' assignment-expression
tree-array ::= '[' tree-array-item [',' tree-array-item]* ']'
tree-array-item ::= expression ':' assignment-expression |
assignment-expression
```

An extension of the syntax allows the declaration of more than one variable in one shot. A comma separates the variable declarations:

```
local-variable-statement ::= "local" local-variable-declaration [
',' local-variable-declaration ]* ';'
```

The local variable points to a new empty tree, pushed into the stack.

- If an expression is present after the local declaration, it is evaluated and the string result is assigned to the new local variable.

- If a constant tree is present after the local declaration, it is assigned to the new local variable. Example:
  ```
  local aVariable = {"a", {"yellow", "red"}, "submarine"};
  ```
  is equivalent to:
  ```
  local aVariable;
  pushItem aVariable = "a";
  pushItem aVariable;
  pushItem aVariable#back = "yellow";
  pushItem aVariable#back = "red";
  pushItem aVariable = "submarine";
  ```
  where `pushItem` means that a new item has to be added in the array owned by `aVariable`, and where `#last` means accessing to the last item of the array.

To assign a reference to another variable, instead of either the result of evaluating an expression or a constant tree, use rather the following declaration statement:
```
local-ref-statement ::= "localref" local-ref-declaration [ ','
local-ref-declaration ]* ';'
local-ref-declaration ::= variable '=' reference
```

In the case of a CODEWORKER version **strictly older than 1.13**, local variables that are declared in the body of a script or in the scope of a function may be accessed further in the scope of functions during their timelife. So a different behaviour may occur with a more recent CODEWORKER interpreter.

This stack management had historical reasons, but it is now obsolete and often reflects an implementation's error. To preserve you from this kind of mistake, a warning may be displayed, so that scripts strictly older than version 1.13 may continue to run. Specify a version strictly older than 1.13 to the command line (option -version) for reclaiming that CODEWORKER checks and generates a warning.

To correct this kind of mistake in old scripts, the variable should be propagated in an argument for functions that refer to it.

To declare a global variable, use the global statement. The declaration of a *global* variable can be specified anywhere in scripts. The first time the declaration of a global variable is encountered, the interpreter registers it as accessible from any point into scripts. The second time the interpreter encounters a global declaration for the variable, the latter remains *global* but its content is cleared.
Note that if a local variable or an attribute of the current node (this) is identical to the name of an existing *global* variable, the global variable remains hidden while the flow of control hasn't left the scope that contains the homonym.

the global declaration statement looks like:
```
global-variable-statement ::= "global" global-variable-declaration [
',' global-variable-declaration ]* ';'
global-variable-declaration ::= variable [ '=' assignment-expression
]?
```

Navigating along branches

It is possible to navigate along a branch of the subtree put into the variable. A branch points to a node of the subtree. The syntax looks generally like:

```
branch ::= variable ['.'  sub-node]*
```

If the branch isn't known before runtime, it may be build during the execution.

Example: while parsing an XML file, each time an XML attribute is encountered, one creates the corresponding attribute into the parse tree. But the name of the attribute is discovered during the parsing. The directive `#evaluateVariable(`*expression*`)` allows doing it. *expression* is evaluated at runtime and provides a branch:

`#evaluateVariable("a.b.c")` will resolve the path *"a.b.c"* at runtime and navigate from *a* to textit*c*.

A node may contain an array of nodes, which are indexed by a key that is a constant string. A branch allows navigating through arrays, and the definitive syntax of branches conforms to:

```
branch ::= "#evaluateVariable" '(' expression ')'
           ::= variable ['.'  sub-node | array-access]*
array-access ::= '[' expression ']'
           ::= '#' ["front" | "back" | "parent"] | "root"]
           ::= '#' '[' integer-expression ']'
```

We see that there are some ways to access an item node of an array or to change how to navigate from nodes to nodes:

- *sub-node* `'[' expression ']'` means that we'll access the node item associated to the string key resulting of the expression's evaluation,

- *sub-node* `'#'` `"front"` means that the first item node of the array is required. If the array is empty, an error occurs.

- *sub-node* `'#'` `"back"` means that the last item node of the array is required. If the array is empty, an error occurs.

- *sub-node* `'#'` `"parent"` means that one comes back up the parent's node of *sub-node*.

- *sub-node* `'#'` `"root"` means that one comes back up the root's node of the tree *sub-node* belongs to.

- *sub-node* `'#'` `'['` <integer-expression> `']'` means that we'll access the node item located at the position given by the evaluation of the expression. The position starts counting to 0. An error is raised if the position is out of bounds.

Assignments

CODEWORKER provides some different ways to put a data into a variable or into the node pointed to by a branch:

- **set** *variable-branch* `['=' | '+=']` *assignment-expression*: the expression is evaluated and the resulting string value is assigned to the variable, or concatenated if the operator `'+='` was required. Keyword `set` may be omitted. The node to assign is supposed existing yet. If not, the assignment is done, but it causes a warning to prevent a spelling mistake on the variable's name.

- **insert** *variable-branch* `[['=' | '+=']` *assignment-expression*`]?` : it works like the `set` assignment, except that it is the preferred mode to add a new node when

the variable doesn't exist yet. If the node already exists, of course it isn't added twice, and the assignment if done as expected. If no assignment is specified after the variable's name, nothing is assigned to the node. So, if the node wasn't existing yet, it contains an empty string. Otherwise, the ancient value isn't changed.

- **ref** *variable* **'='** *existing-variable-or-branch* : the variable to assign will refer to an existing node. Inspecting the variable will cause inspecting the referenced existing node. If the referenced node doesn't exist, an error occurs. If you apply the reference to a variable that already refers a node, this link is broken instead of propagating the reference to the referred node. This operator is very useful during the decoration of the parse tree, and leads to transform the tree as a freely-oriented graph.

  Be careful not to keep a reference to a local variable once the flow of execution has left its scope: the local variable is deleted, and so, the reference points to a corrupted part of the memory.

  If you intend to assign a reference to a variable into a function and that the variable is passed by parameter, don't forget to take the reference parameter mode:

  ```
  function badFunction(myVar :  node) {
      ...
      // myVar will keep up a reference to aNode
      // up to the end of the function:
      ref myVar = aNode;
      ...
      // myVar is passed as variable, so the
      // reference is cancelled once the function is left!
  }

  // To keep the reference after leaving the function, change the
  parameter // mode to reference:  function goodFunction(myVar :
  reference) {
      ...
      // myVar will keep up a reference to aNode
      // up to the end of the function:
      ref myVar = aNode;
      ...
      // myVar is passed as reference, so the
      // reference is kept once the function is left!
  }
  ```

- **setall** *variable-branch* **'='** *existing-variable-or-branch* : value, attributes and array of the variable to assign are purged, and the subtree, to which the existing variable points, is copied integrally to the node to assign.

- **merge** *variable-branch* **'='** *existing-variable-or-branch* : the subtree, to which the existing variable points, is copied integrally to the node to assign, preserving the attributes and the arrays of the assigned node, which are updated or completed.

- **pushItem** *variable-branch* [**'='** *expression*]? : a new item node is added at the end of the variable's array, whose key is worth its position, starting at 0. If the expression exists, then after evaluating it, the result is assigned to the item node as a value. If no array was previously existing, the item becomes its first component.

4.2.4 Expressions

Presentation

The BNF representation of an expression looks like:
```
expression ::= boolean-expr | ternary-expr
boolean-expr ::= comparison-expr [boolean-op comparison-expr]
boolean-op ::= '&' | '&&' | '|' | '||' | '^' | '^'
ternary-expr ::= comparison-expr '?'  expression ':'  expression
comparison-expr ::= concatenation-expr [comparison-op
concatenation-expr | "in" constant-set]
constant-set ::= '{' constant-string [',' constant-string]* '}'
comparison-op ::= '<' | '<=' | '==' | '=' | '!=' | '<>' | '>' | '>='
concatenation-expr ::= stdliteral-expr ['+' stdliteral-expr]*
stdliteral-expr ::= literal-expr
             ::= '$' arithmetic-expr '$'
literal-expr ::= constant-string | number
             ::= "true" | "false"
             ::= '(' expression ')'
             ::= '!'  literal-expr
             ::= preprocessor-expr
             ::= function-call
             ::= variable-or-branch

 arithmetic-expr ::= comparith-expr [boolean-op comparith-expr]*
comparith-expr ::= sum-expr [comparison-op sum-expr]
sum-expr ::= shift-expr [['+' | '-'] shift-expr]*
shift-expr ::= factor-expr [["«" | "»"] factor-expr]*
factor-expr ::= literal-expr [['*' | '/' | '%'] literal-expr]*
unary-expr ::= literal-expr ["++" | "-"]
literal-expr ::= string | variable-expr | number | unary-expr
             ::= ' ' literal-expr
preprocessor-expr ::= '#' ["LINE" | "FILE"]
```
where:

- The boolean operators `&` or `&&` resolve the logical AND.

- The boolean operators `|` or `||` resolve the logical OR.

- The comparison operators do the classical resolution, working on the lexicographical order. Note that *different of* may be written `!=` or `<>` and that the equality is written`=` or `==`. If the comparison succeeds, it returns `"true"`, otherwise, it returns an empty string. See section 4.2.4 to discover an escape mode for writing arithmetic comparisons.

- A special comparison operator **in** checks whether the left-hand side member belongs to a set of constant strings or not.
  Example:
  sHTMLTag **in {** 'i', "kbd" **}**
  returns `true` if *sHTMLTag* is worth 'i' or "kbd" and `false` in all other cases.

- The operator + serves to concatenate two string. Becareful not to use this operator to ask for an arithmetic addition! See the function `add()` to do that or the section 4.2.4 to discover an escape mode for writing arithmetic operators.

- A constant string must be written between double quotes and escape characters are expected as in C, starting with a back-slash.

- The syntax of a number is the one admitted commonly. The number is then converted to a string.

- Constant `true` is worth the constant string `"true"`, and `false` is worth the constant string `"false"`.

- expressions with parentheses are allowed.

- The unary `!` operator resolves the logical NOT, such as an empty value gives `"true"` and any kind of other value gives an empty string.

- Function calls are described into the section 4.2.5.

- The escape mode for arithmetic expression is set/unset via the '**$**' symbols and allows interpreting arithmetic/comparison operators as usual ('`+`' as an addition, '`<`' as a numerical comparison). See section 4.2.4 for more information about this escape mode.

- some arithmetic operators enable to handles bits: the unary bitwise operator **~** and the shift operators **«** (left shifting) and **»** (right shifting).

- The preprocessor expressions give information about the source script:

  - **#FILE**: returns the file name of the script, or an empty string if the source script wasn't coming from a file,

  - **#LINE**: returns the line number where the directive is located into the source script,

Arithmetic expressions

The classical syntax of the interpreter forces expressions to work on sequences of characters. So, comparison operators apply the lexicographical order and the '`+`' operator concatenates two strings and the '`*`' operator doesn't exist.

Of course, it exists some functions to handle strings as number and to execute an arithmetic operation (the '`add()`' or '`mult()`' functions for instance) or a comparison (the '`isPositive()`' or '`inf()`' functions for instance).

However, it appears clearly more convenient to write arithmetic operations and comparisons in a natural way, using operators instead of the corresponding functions. So, CODEWORKER provides an escape mode that draws its inspiration from *LaTeX* to express mathematical formulas: the arithmetic expression are delimited by the symbol '**$**'.

Example:

```
local a = 11;
local b = 7;
traceLine("Classical mode = '"
+ inf(add(mult(5, a), 3), sub(mult(a, a), mult(b, b))) + "'");
traceLine("Escape mode = '" + $5*a + 3 < a*a - b*b$ + "'");
```

Output:

```
Classical mode = 'true'
Escape mode = 'true'
```

4.2.5 Common statements

The 'if' statement

The BNF representation of the `while` statement is:
```
if-statement ::= "if" expression then-statement ["else"
else-statement]?
```

The `if` statement evaluates the expression following immediately. The expression must be of arithmetic, text, variable or condition type. In both forms of the `if` syntax, if the expression evaluates to a nonempty string, the statement dependent on the evaluation is executed; otherwise, it is skipped.

In the `if...else` syntax, the second statement is executed if the result of evaluating the expression is an empty string. The `else` clause of an `if...else` statement is associated with the closest previous `if` statement that does not have a corresponding `else` statement.

The 'while'/'do' statements

The BNF representation of the `while` statement is:
```
while_statement ::= "while" expression statement
```

The `while` statement lets you repeat a statement or compound statement as long as a specified *expression* becomes an empty string. The `expression` in a `while` statement is evaluated before the body of the loop is executed. Therefore, the body of the loop may be never executed. If `expression` returns an empty string, the `while` statement terminates and control passes to the next statement in the program. If `expression` is non-empty, the process is repeated. The `while` statement can also terminate when a **break**, or `return` statement is executed within the statement body. When a **continue** statement is encountered, the control breaks the flow and jumps to the evaluation of the `expression`.

Note that the **break** and **continue** statements apply to the first loop statement (`foreach`/`forfile`/`select`, `do`/`while`) they encounter while leaving instruction blocks.

The BNF representation of the `do` statement is:
```
do_statement ::= "do" statement "while" expression ';'
```

The `do-while` statement lets you repeat a statement or compound statement until a specified `expression` becomes an empty string. The *expression* in a `do-while` statement is evaluated after the body of the loop is executed. Therefore, the body of the loop is always executed at least once. If *expression* returns an empty string, the `do-while` statement terminates and control passes to the next statement in the program. If *expression* is non-empty, the process is repeated. The `do-while` statement can also terminate when a `break`, or `return` statement is executed within the statement body. When a `continue` statement is encountered, control is transferred to the evaluation of the *expression*.

The 'switch' statement

The BNF representation of this statement is:
```
switch_statement ::= "switch" '(' expression ')' '{'
(label_declaration)* ("default" ':'  statement)?  '}'
label_declaration ::= ["case" | "start"] constant_string ':'
statement
```

The `switch` statement allows selection among multiple sections of code, depending on the value of an expression. The expression enclosed in parentheses, the controlling expression, must be of string type.

The `switch` statement causes an unconditional jump to, into, or past the statement that is the switch

body, depending on the value of the controlling expression, the constant string values of the `case` or `start` labels, and the presence or absence of a `default` label. The switch body is normally a compound statement (although this is not a syntactic requirement). Usually, some of the statements in the `switch` body are labeled with `case` labels or with `start` labels or with the `default` label. The default label can appear only once.

The constant-string in the `case` label is compared for equality with the controlling expression. The constant-string in the `start` label is compared for equality with the first characters of the controlling expression. In a given `switch` statement, no two constant strings in `start` or `case` statements can evaluate to the same value.

The `switch` statement behaviour depends on how the controlling expression matches with labels. If a `case` label exactly matches with the controlling expression, control is transferred to the statement following that label. If failed, `start` labels are iterated into the lexicographical order, and the control is transferred to the statement following the first label that matches with the beginning of the controlling expression. If failed, control is transferred to the `default` statement or, if not present, an error is thrown.

A `switch` statement can be nested. In such cases, `case` or `start` or `default` labels associate with the most deeply nested `switch` statements that enclose them.

Control is not impeded by `case` or `start` or `default` labels. To stop execution at the end of a part of the compound statement, insert a `break` statement. This transfers control to the statement after the `switch` statement.

The 'foreach' statement

The BNF representation of this statement is:
```
foreach_statement ::= "foreach" iterator "in" [direction]?
              [sorted_declaration]?  [cascading_declaration]?
list-node body_statement
direction ::= "reverse"
sorted_declaration ::= "sorted" ["no_case"]?  ["by_value"]?
cascading_declaration ::= "cascading" ["first" | "last"]?
```

A *foreach* statement iterates all items of the list owned by node *list-node*. The *iterator* refers to the current item of the list, and the body statement is executed on it.

Items are iterated either in the order of entrance, or in alphabetical order if option `sorted` is set. The sort operates on keys, except if the option `by_value` is set. The order is inverted if option `reverse` was chosen. To ignore the case, these options must be followed by `no_case`. If not, uppercase letters are considered as smaller than any lowercase letter.

```
   // file "Documentation/ForeachSampleSorted.cws":
local list;
insert list["silverware"] = "tea spoon";
insert list["Mountain"] = "Everest";
insert list["SilverWare"] = "Tea Spoon";
insert list["Boat"] = "Titanic";
insert list["acrobat"] = "Circus";

traceLine("Sorted list in a classical order:");
foreach i in sorted list {
    traceLine("\t" + key(i));
}
```

```
        traceLine("Note that uppercases are listed before lowercases."
+ endl());

        traceLine("Sorted list where the case is ignored:");
        foreach i in sorted no_case list {
            traceLine("\t" + key(i));
        }

        traceLine("Reverse sorted list:");
        foreach i in reverse sorted list {
            traceLine("\t" + key(i));
        }

        traceLine("Reverse sorted list where the case is ignored:");
        foreach i in reverse sorted no_case list {
            traceLine("\t" + key(i));
        }
```

Output:

```
Sorted list in a classical order:
    Boat
    Mountain
    SilverWare
    acrobat
    silverware
Note that uppercases are listed before lowercases.

Sorted list where the case is ignored:
    acrobat
    Boat
    Mountain
    SilverWare
    silverware
Reverse sorted list:
    silverware
    acrobat
    SilverWare
    Mountain
    Boat
Reverse sorted list where the case is ignored:
    silverware
    SilverWare
    Mountain
    Boat
    acrobat
```

Control may not be sequential into the body statement. `break` and `return` enable exiting definitely the loop, and `continue` transfers the control to the head of the `foreach` statement for the next iteration.

Option `cascading` allows propagating `foreach` on item nodes. The way it works is illustrated by an example:

```
foreach i in cascading myObjectModeling.packages ...
```

At the beginning, `i` points to `myObjectModeling.packages#front` and the body is executed. Before iterating i to the next item, the `foreach` checks whether the item node `myObjectModeling.packages#front` owns attribute `packages` or not. If yes, it applies recursively `foreach` on `myObjectModeling.packages#front.packages`.

Option `cascading` avoids writing the following code:

```
function propagateOnPackages(myPackage : node)
foreach i in myPackage
// my code to apply on this package
if existVariable(myPackages.packages)
propagateOnPackages(myPackages.packages);


propagateOnPackages(myObjectModeling.packages);
```

Option `cascading` offers two behaviours:

- `first` means that the item is cascaded before running the body,

```
 // file "Documentation/ForeachSampleFirst.cws":
local myObjectModeling;
insert myObjectModeling.packages["Massif"] = "...";
local myPackage;
ref myPackage = myObjectModeling.packages["Massif"];
insert myPackage.packages["Alps"] = "...";
insert myPackage.packages["Himalaya"] = "...";
insert myPackage.packages["Rock Mountains"] = "...";
insert myObjectModeling.packages["Silverware"] = "...";
ref myPackage = myObjectModeling.packages["Silverware"];
insert myPackage.packages["Spoon"] = "...";
insert myPackage.packages["Fork"] = "...";
insert myPackage.packages["Knife"] = "...";

foreach i in cascading first myObjectModeling.packages {
    traceLine("\t" + key(i));
}
```

  Output:

```
Alps
Himalaya
Rock Mountains
Massif
Spoon
Fork
Knife
Silverware
```

- `last` is the default behaviour, as seen in previous examples, and

```
 // file "Documentation/ForeachSampleLast.cws":
local myObjectModeling;
insert myObjectModeling.packages["Massif"] = "...";
local myPackage;
ref myPackage = myObjectModeling.packages["Massif"];
insert myPackage.packages["Alps"] = "...";
```

```
    insert myPackage.packages["Himalaya"] = "...";
    insert myPackage.packages["Rock Mountains"] = "...";
    insert myObjectModeling.packages["Silverware"] = "...";
    ref myPackage = myObjectModeling.packages["Silverware"];
    insert myPackage.packages["Spoon"] = "...";
    insert myPackage.packages["Fork"] = "...";
    insert myPackage.packages["Knife"] = "...";


    foreach i in cascading last myObjectModeling.packages {
        traceLine("\t" + key(i));
    }
```
Output:
```
    Massif
    Alps
    Himalaya
    Rock Mountains
    Silverware
    Spoon
    Fork
    Knife
```
propagates the `foreach` on the current item after executing the body.

The 'forfile' statement

The BNF representation of this statement is:

forfile_statement ::= **"forfile"** *iterator* **"in"** [*sorted_declaration*]?
[*cascading_declaration*]? *file-pattern body_statement*
sorted_declaration ::= **"sorted"** [**"no_case"**]?
cascading_declaration ::= **"cascading"** [**"first"** | **"last"**]?

A *forfile* statement iterates the name of all files that verify the filter *file-pattern*. The *iterator* refers to the current item of the list composed of retained file names, and the body statement is executed on it. Note that the file pattern may begin with a path, which cannot contain jocker characters ('*' and '?').

Like for the `foreach` statement, items are iterated either in the order of entrance, or in alphabetical order of keys if option `sorted` is set. To ignore the case, the option must be followed by `no_case`. If not, uppercase letters are considered as smaller than any lowercase letter.

Control may not be sequential into the body statement. `break` and `return` enable exiting definitely the loop, and `continue` transfers the control to the head of the `forfile` statement for the next iteration.

The option `cascading` allows propagating `forfile` on directories recursively. The way it works is illustrated by an example:

```
    // file "Documentation/ForfileSample.cws":
    local iIndex = 0;
    forfile i in cascading "*.html" {
        if $findString(i, "manual_") < 0$ &&
           $findString(i, "Bugs") < 0$ {
              traceLine(i);
        }
        // if too long, stop the iteration
        if $iIndex > 15$ break;
```

Chapter 4. The scripting language

```
        increment(iIndex);
    }
```

Output:

```
cs/DOTNET.html
cs/tests/data/MatchingTest/example.csv.html
Documentation/LastChanges.html
java/JAVAAPI.html
java/data/MatchingTest/example.csv.html
Scripts/Tutorial/GettingStarted/defaultDocumentation.html
WebSite/AllDownloads.html
WebSite/examples/basicInformation.html
WebSite/highlighting/basicInformation.html
WebSite/repository/highlighting.html
WebSite/repository/JEdit/Entity.java.cwt.html
WebSite/serewin/ExempleIllustre.html
WebSite/tutorials/DesignSpecificModeling/tutorial.html
WebSite/tutorials/DesignSpecificModeling/highlighting/demo.cws.html
WebSite/tutorials/overview/tinyDSL_spec.html
WebSite/tutorials/overview/scripts2HTML/CodeWorker_grammar.html
```

At the beginning, i points to the first HTML file of the current directory and the body is executed. Before iterating i to the next item, the `forfile` checks whether the directory of the current file owns subfolders or not. If yes, it applies recursively `forfile` on subfolders.

Option `cascading` offers two behaviours:

- `first` means that the subfolders are visited before running the body,

- `last` is the default behaviour, as seen in previous examples, and propagates the `forfile` on the subfolder after executing the body.

The 'select' statement

The BNF representation of this statement is:
select_statement ::= **"select"** *iterator* **"in"** [*sorted_declaration*]?
node-motif body_statement
sorted_declaration ::= **"sorted"** *first-key* [, *other-key*]*
first-key ::= *branch*
other-key ::= *branch*

A *select* statement iterates a list of nodes that match a motif expression. The *iterator* refers to the current item of the list composed of retained nodes, and the body statement is executed on it.

```
   // file "Documentation/SelectSample.cws":
   local a;
   pushItem a.b;
   pushItem a.b#back.c = "01";
   pushItem a.b#back.c = "02";
   pushItem a.b#back.c = "03";
   pushItem a.b;
   pushItem a.b#back.c = "11";
   pushItem a.b#back.c = "12";
```

```
        pushItem a.b#back.c = "13";
        pushItem a.b;
        pushItem a.b#back.c = "21";
        pushItem a.b#back.c = "22";
        pushItem a.b#back.c = "23";
        select i in a.b[].c[] {
            traceLine("i = "+ i);
        }
```

Output:

```
i = 01
i = 02
i = 03
i = 11
i = 12
i = 13
i = 21
i = 22
i = 23
```

Like for the `foreach` statement, items are iterated either in the order of entrance, or according to the sorting result if the option `sorted` is set.

Control may not be sequential into the body statement. `break` and `return` enable exiting definitely the loop, and `continue` transfers the control to the head of the `select` statement for the next iteration.

The 'try'/'catch' statement

The BNF representation of this statement is:
try-catch-statement ::= **"try"** *try-statement* **"catch"** *'('error_message_variable')' catch-statement*

Error handling is implemented by using the `try`, `catch`, and `error` keyword. With error handling, your program can communicate unexpected events to a higher execution context that is better able to recover from such abnormal events. These errors are handled by code that is outside the normal flow of control.

The compound statement after the `try` clause is the guarded section of code. An error is thrown (or raised) when command `error`(*message-text*) is called or when CODEWORKER encounters an internal error. The compound statement after the `catch` clause is the error handler, and catches (handles) the error thrown. The `catch` clause statement indicates the name of the variable that must receive the error message.

The 'exit' statement

The BNF representation of this statement is:
exit_statement ::= **"exit"** *integer-expression* **";"**

A *exit* statement leaves the application and returns an error code, given by the *integer-expression*.

Example:
```
exit -1;
```

4.2.6 User-defined functions

The BNF representation of a user-defined function to implement is:
```
user-function ::= classical-function-definition |
template-function-definition
classical-function-definition ::= classical-function-prototype
compound-statement
classical-function-prototype ::= "function" function-name '('
parameters ')'
template-function-definition ::= see the next section, 4.2.6, for
more information
parameters ::= parameter [',' parameter]*
parameter ::= argument [':'  parameter-mode [':'  default-value]?
]?
parameter-mode ::= "value" | "node" | "reference" | "index"
default-value ::= "project" | "this" | "null" | "true" | "false" |
constant-string
```

The scripting language allows the user implementing its own functions. Parameters may be passed to the body of the function. A value may be returned by the function and, if so, the return type is necessary a sequence of characters. Of course, functions manage their own stack, and so, accept recursive calls.

An argument may have a default value if the parameter is missing in a call. All following arguments must then have default values too. A **node** argument can't have a constant string as a default argument, but it can be worth a global variable.

Parameters and return value

Arguments passed by parameter must be chosen among the following modes:

- `value`: if the mode of argument is omitted, this is the default mode ; it requires a sequence of characters (a value of node, a constant string or the result of a expression),

- `node`: a node is passed and it may be changed or inspected in the body. The scope of a reference assignment is limited to the scope of the function: once the function is left, the variable receives the value of the referenced node. It is explained by the fact that the parameter is a new local variable, which refers to the node passed as argument. So, a reference assignment is applied on the local variable only.

- `iterator`: the iterator of a `foreach` statement is expected, for applying iterator functions on the argument (*first()* for instance). Not really useful and `node` is now sufficient.

- `reference`: a node is passed and it may be changed or expected in the body. On the contrary of `variable` mode, a reference assignment is propagated outside the scope of the function.

If you have omitted to return a value from a function, it returns an empty string ; in that case, you expects to call this function as a procedure and the result isn't exploited. The special procedure `nop` takes a function call as parameter and allows executing the function and ignoring the result. It isn't compulsory to use `nop` for calling a function as a procedure. As in C or C++, you can type the function call followed by a semi-colon and the result is lost.

It exists two possibilities for returning a value:

- to populate an internal local variable whose name is the same as the function name,

---

- to use the `return` statement, followed by the expression to evaluate,

If you wish to execute a particular process in any case before leaving a function and:

- it exists more than one controlling sequence to leave,

- some errors may be raised,

The 'finally' statement

the statement `finally` warrants you that the block of instructions that follows the keyword will be systematically executed before leaving. This declaration may be placed anywhere into the body of the function. Its syntax conforms to:

finally-statement ::= **"finally"** *compound-statement*

Example:

```
    // file "Documentation/FinallySample.cws":
1  function f(v :  value) {
2      traceLine("BEGIN f(v)");
3      finally {
4          traceLine("END f(v)");
5      }
6      // the body of the function, with more than
7      // one way to exit the function, for example:
8      if !v return "empty";
9      if v == "1" return "first";
10     if v == "2" return "second";
11     if v == "3" return "third";
12     return "other";
13 }
14
15 traceLine("...f(1) has been executed and returned '" + f(1) +
"'");
```

LINE 3: the `finally` statement is put anywhere in the body,
LINE 4: this statement will be executed while exiting the function, even if an exception was raised,

Output:

```
BEGIN f(v)
END f(v)
...f(1) has been executed and returned 'first'
```

Unusual function declarations

It may arrive that a function prototype must be declared before being implemented, because of a cross-reference with another function for instance. The scripting language offers the forward declaration to answer this need. To do that, the prototype of the function is written, preceded by the `declare` keyword:

forward-declaration ::= **"declare"** *function-prototype* ';'

If the body of the function must be implemented in another library and into C++ for example, the prototype of the function is preceded by the `external` keyword (see section 5.1):

```
external-declaration ::= "external" function-prototype ';'
```

Template functions

CODEWORKER proposes a special category of functions called *template functions*. Because of CODEWORKER doesn't provide a typed scripting language, *template* hasn't to be understood as it is commonly exploited in C++ for instance.

A *template function* represents a set of functions with the same prototype, except the *dispatching constant*. The *dispatching constant* is a constant string that extends that name of the function. These functions *instantiate* the *template function* for a particular *dispatching constant*. Each instantiated function implements its own body.

The BNF representation of a template function to implement is:

```
template-function-definition ::= instantiated-function-definition |
generic-function-definition
instantiated-function-definition ::= instantiated-function-prototype
compound-statement
instantiated-function-prototype ::= "function" function-name '<'
dispatching-constant '>' '(' parameters ')'
dispatching-constant ::= a constant string between double quotes
generic-function-definition ::= generic-function-prototype
[compound-statement | template-based-body]
generic-function-prototype ::= "function" function-name '<'
generic-key '>' '(' parameters ')'
generic-key ::= an identifier that matches any dispatching constant
with no attached prototype
template-based-body ::= "{{" template-based-script "}}"
template-based-script ::= a piece of template-based script
describing the generic implementation
```

A call to a *template function* requires to provide a *dispatching expression* to determine the *dispatching constant*. The *dispatching expression* will be evaluated during the execution and CODEWORKER will resolve what *instantiated function* of this template to call: the result of the *dispatching expression* must match with the *dispatching constant* of the *instantiated function*. The BNF representation of a call to a template function is:

```
instantiated-function-call ::= function-name '<'
dispatching-expression '>' '(' parameters ')'
parameters ::= expression [',' expression]*
```

Note that a *dispatching constant* may be empty and such an instantiated function can be called as a classical function. In fact, classical functions are considered as *instantiated functions* where the *dispatching constant* is empty.

template functions bring generic programming in the language: let imagine that we need function `getType(myType :   node)`, to decline for every language we could have to generate (C++, Java, ...). Normally, you'll write the following lines to recover the type depending on the language for which you are producing the source code:

```
if doc_language == "C++" {
sType = getCppType(myParameterType);
} else if doc_language == "JAVA" {
sType = getJAVAType(myParameterType);
} else {
error("unrecognized language '" + doc_language + "'");
}
```

Thanks to the template functions, you may replace the precedent lines by the next one:

```
sType = getType<doc_language>(myParameterType);
```

with:

```
function getType<"JAVA">(myType : node) {
... // implementation for returning a Java type
}

function getType<"C++">(myType : node) {
... // implementation for returning a C++ type
}
```

During the execution, the function `getType<`**T**`>(myType :  node)` resolves on what instantiated function it has to dispatch: either `getType<`**"JAVA"**`>(myType :  node)` or `getType<`**"C++"**`>(myType :  node)`, depending on what value is assigned to variable *doc_language*.

Trying to call an instantiated function that doesn't exist, raises an error at runtime. However, one might imagine an implementation by default. For instance:

```
function getType<T>(myType : node) {
... // common implementation for any unrecognized language
}
```

For those that know generic programming with C++ templates, here is a classical example of using template functions:

```
function f<1>() { return 1; }
function f<N>() { return $N*f<$N - 1$>()$; }
local f10 = f<10>();
if $f10 != 3628800$ error("10! should be worth 3628800");
traceLine("10! = " + f10);
```

Output:

```
10!  = 3628800
```

To provide more flexibility in the implementation of the template function, depending on the generic key **<T>**, the body admits a *template-based* script to implement the source code of the function. The specialization of the function for a given template instantiation key is then resolved at runtime.

Example:
The template function `f` inserts a new attribute in a tree node. The attribute has the name passed to the generic key for instantiation, and the value of the instantiation key is assigned to the new attribute. Then, the function calls itself recursively on the instantiation key without the last character.

For instance, the source code of `f<"field">` should be:
```
function f<"field">(x :  node) {
     insert x.field = "field";
     f<"fiel">(x); // cut the last character
}
```
Code:
```
//a synonym of f<"">(x :  node), terminal condition for recusive
calls
function f(x :  node) {/*does nothing*/}
function f<T>(x :  node) {{
     // '{{' announces a template-based script, which
     // will generate the correct implementation during the
instantiation
     insert x.@T@ = "@T@";
     f<"@T.rsubString(1)@">(x);
@
     // '}}' announces the end of the template-based script
}}
f<"field">(project);
traceObject(project);
```
Output:
```
Tracing variable 'project':
     field = "field"
     fiel = "fiel"
     fie = "fie"
     fi = "fi"
     f = "f"
End of variable's trace 'project'.
```

Methods

For more readability, syntactical facilities are offered to call functions on a node as if this function was a method of the node. For example, it is possible to call function `leftString` on the node *a* like this: `a.leftString(2)`, instead of the classical functional form: `leftString(a, 2)`.

The rule is that every function (user-defined included) whose first argument is passed either by `value` or by `node` or by `index` (but never by `reference`) can propose a method call.

In that case, the method call applies on the first argument, which has to be a node. The BNF representation of a method call is:
```
method-call ::= variable '.'  function-name '(' parameters ')'
parameters ::= expression [',' expression]*
```
where `parameters` have missed the first argument of the function called *function-name*.

It exists some exceptions where the method doesn't apply to the first argument:

- `findElement` applies on the second argument,

- `replaceString` applies on the third argument,

The following methods offer a synonym to the function name:

- `empty` is a synonym as a method of the function `isEmpty`,

- length is a synonym for the function lengthString,

- size is a synonym for the function getArraySize,

The 'readonly' hook

The BNF representation of this statement is:
readonlyHook-statement ::= **"readonlyHook"** *'(' filename ')'*
compound-statement

The token *filename* is the argument name that the user chooses for passing the name of the file to the body of the hook.

This special function allows implementing a hook that will be called each time a read-only file will be encountered while generating the output file through the generate or expand instruction.

Limitations: only one declaration of this hook is authorized, and it can't be declared inside a parsing or pattern script.

Example:

Common usage: file to generate has to be checked out from a source code control system (see system command to run executables).

```
readonlyHook(sFilename) {
  if !getProperty("SSProjectFolder") || !getProperty("SSWorkingFolder")
|| !getProperty("SSExecutablePath") || !getProperty("SSArchiveDir")
{
    traceLine("WARNING: properties 'SSProjectFolder' and
'SSWorkingFolder' and 'SSExecutablePath' and 'SSArchiveDir' should
be passed to the command line for checking out read-only files from
Source Safe");
  } else {
    if startString(sFilename, getProperty("SSWorkingFolder")) {
      local sourceSafe;
      insert sourceSafe.fileName = sFilename;
      generate("SourceSafe.cwt", sourceSafe, getEnv("TMP") +
"/SourceSafe.bat");
      if sourceSafe.isOk {
        putEnv("SSDIR", getProperty("SSArchiveDir"));
        traceLine("checking out '" + sFilename + "' from Source Safe
archive '" + getProperty("SSArchiveDir") + "'");
        local sFailed = system(getEnv("TMP") + "/SourceSafe.bat");
        if sFailed {
          traceLine("Check out failed:  '" + sFailed + "'");
        }
      }
    } else {
      traceLine("Unable to check out '" + sFilename + "':  working
folder starting with '" + getProperty("SSWorkingFolder") + "'
expected");
    }
  }
}
```

The 'write file' hook

This special function allows implementing a hook that will be called just before writing a file, after ending a text generation process such as expanding or generating or translating text.

It is very important to notice that it returns a boolean value. A `true` value means that the generated text must be written into the file. A `false` boolean value means that the generated text doesn't have to be written into the file.

CODEWORKER always interprets not returning a value explicitly of a function, as returning an empty string. If you forget to return a value, the generated text will not be written into the file!

The BNF representation of this statement is:
```
writefileHook-statement ::= "writefileHook" '(' filename ','
position ',' creation ')' compound-statement
```

Argument	Type	Description
filename	`string`	The argument name that the user chooses for passing the file name to the body of the hook.
position	`int`	The argument name that the user chooses for passing a position where a difference occurs between the new generated version of the file and the precedent one. If the files don't have the same size, the position is worth -1.
creation	`boolean`	The argument name that the user chooses for passing whether the file is created or updated. The argument is worth `true` if the file doesn't exist yet.

Limitations: only one declaration of this hook is authorized, and it can't be declared inside a parsing or pattern script.

Example:

```
writefileHook(sFilename, iPosition, bCreation) {
    if bCreation {
        traceLine("Creating file '" + sFilename + "'!");
    } else {
        traceLine("Updating file '" + sFilename + "', difference at "
+ iPosition + "!");
    }
    return true;
}
```

The 'step into' hook

This special function is automatically called before that the extended BNF engine resolves the production rule of a BNF non-terminal. Combined with `stepoutHook()`, it is very useful for trace and debug tasks.

This hook can be implemented in parse scripts only.

The BNF representation of this statement is:
```
stepintoHook-statement ::= "stepintoHook" '(' sClauseName ','
localScope ')' compound-statement
```

Argument	Type	Description
sClauseName	`string`	The name of the *non-terminal*.
localScope	`tree`	The scope of parameters used into the production rule.

The 'step out' hook

This special function is automatically called once the extended BNF engine has finished the resolution of a BNF non-terminal. Combined with `stepintoHook()`, it is very useful for trace and debug tasks.

This hook can be implemented in parse scripts only.

The BNF representation of this statement is:

```
stepoutHook-statement ::= "stepoutHook" '(' sClauseName ','
localScope ',' bSuccess ')' compound-statement
```

Argument	Type	Description
sClauseName	`string`	The name of the *non-terminal*.
localScope	`tree`	The scope of local variables and parameters used into the production rule.
bSuccess	`boolean`	Whether the resolution of the production rule has succeeded or not.

4.2.7 Statement's modifiers

A statement's modifier is a directive that stands just before a statement, meaning an instruction or a compound statement.

This directive operates some actions in the scope of the statement and then restores the behaviour as being before.

This action may be:

- to measure the time that is consumed by the execution of the statement,

- to redirect into a variable all messages intended to the console during the execution of the statement,

- to push a new `project` parse tree,

- to change the output file during the execution of the statement, while generating text,

- to redirect the output stream into a variable during the execution of the statement, while generating text,

- to change the output file during the execution of the statement, while generating text, and to apply an expansion mode on it,

Statement's modifier 'delay'

This keyword stands just before an instruction or a compound statement. It executes the statement and then, it measures the time it has consumed.

Function `getLastDelay` (4.3.95) gives you the last measured duration.

Example:

```
local list;
local iIndex = 4;
delay while isPositive(decrement(iIndex)) {
pushItem list = "element " + iIndex;
traceLine("creating node '" + list#back + "'");
}
traceLine("time of execution = " + getLastDelay() + " seconds");
```

Output:

```
creating node 'element 3'
creating node 'element 2'
creating node 'element 1'
time of execution = 0.000041625402111162173 seconds
```

Statement modifier 'quiet'

This keyword stands just before an instruction or a compound statement. It executes the statement and all messages intended to the console are concatenated into a string, instead of being displayed. The variable that receives the concatenation of messages is specified after the `quiet` keyword.

The BNF representation of the `quiet` statement modifier looks like:
quiet_modifier ::= **"quiet"** '(' *variable* ')' *statement*

Note that the variable must have been declared before, as a local one or as an attribute of the parse tree. If this variable doesn't exist while executing the statement, an error is raised.

Statement modifier 'new project'

This keyword stands just before an instruction or a compound statement. A new `project` parse tree is created, which is empty and that replaces temporarily the current one. The statement is executed and, once the controlling sequence leaves the statement, the temporary parse tree is removed, and the precedent `project` comes back as the current one.

The BNF representation of the `new_project` statement modifier looks like:
new_project_modifier ::= **"new_project"** *statement*

This statement modifier is useful to handle a task that doesn't have to interact with the main parse tree.

Statement modifier 'file as standard input'

This keyword stands just before an instruction or a compound statement. A new standard input is opened for reading data. Generally, the keyboard is the standard input, but here, it will be the content of a file that is passed to the argument *filename*. Once the execution of the statement has completed, the precedent standard input comes back.

The BNF representation of the `file_as_standard_input` statement's modifier looks like:
`file_as_standard_input_modifier ::= `**`"file_as_standard_input" '('`**` filename `**`')'`**` statement`

This statement modifier is useful to replay a sequence of commands for the debugger or to drive the standard input from an external module that puts its instructions into a file for a batch mode or anything else.

Statement modifier 'string as standard input'

This keyword stands just before an instruction or a compound statement. A new standard input is opened for reading data. Generally, the keyboard is the standard input, but here, it will be the content of the string that is passed to argument. Once the execution of the statement has completed, the precedent standard input comes back.

The BNF representation of the `string_as_standard_input` statement's modifier looks like:
`string_as_standard_input_modifier ::= `**`"string_as_standard_input" '('`**` expression `**`')'`**` statement`

The standard input is the result of evaluating *expression*.

This statement modifier is useful to drive the standard input of CODEWORKER from an external module, such as a JNI library or an external C++ application (see chapter 4.6.38).

Statement modifier 'parsed file'

This keyword stands just before an instruction or a compound statement that belongs to a *parsing/translation script* exclusively. A new input file is opened for source scanning, and replaces temporarily the precedent during the execution of the statement.The statement is executed and, once the controlling sequence leaves the statement, the input file is closed properly and the precedent one comes back.

The BNF representation of the `parsed_file` statement modifier looks like:
`parsed_file_modifier ::= `**`"parsed_file" '('`**` filename `**`')'`**` statement`

The token *filename* is an expression that is evaluated to give the name of the input file.

This statement modifier is useful to handle a task that must redirect the text to parse into another input file. An example could be to emulate the C++ preprocessing on `#include` directives.

Statement modifier 'parsed string'

This keyword stands just before an instruction or a compound statement that belongs to a *parsing/translation script* exclusively. The result of an expression is taken as the source to scan, and replaces temporarily the precedent input during the execution of the statement.The statement is executed and, once the controlling sequence leaves the statement the precedent input comes back.

The BNF representation of the `parsed_string` statement modifier looks like:
`parsed_string_modifier ::= `**`"parsed_string" '('`**` expression `**`')'`**
`statement`

The token *fexpression* is an expression that is evaluated to give the text to scan.

This statement modifier is useful to handle a task that must temporary parse a string.

Statement modifier 'generated file'

This keyword stands just before an instruction or a compound statement that belongs to a *pattern script* exclusively. A new output file is opened for source code generation, preserving protected areas as usually, and replaces temporarily the current one during the execution of the statement. The statement is executed and, once the controlling sequence leaves the statement, the output file is closed properly and the precedent one takes its place.

The BNF representation of the `generated_file` statement modifier looks like:
generated_file_modifier ::= **"generated_file"** '(' *filename* ')'
statement

The token *filename* is an expression that is evaluated to give the name of the output file.

This statement modifier is useful to handle a task that must redirect the generated text into another output file. An example could be to split an HTML text to generate into a few files for implementing a frame set.

Statement modifier 'generated string'

This keyword stands just before an instruction or a compound statement that belongs to a *pattern script* exclusively. The output stream is redirected into a variable that replaces temporarily the current output stream during the execution of the statement. The statement is executed and, once the controlling sequence leaves the statement, the variable is populated with the content of the output produced during this scope and the precedent output stream takes its place.

The BNF representation of the `generated_string` statement modifier looks like:
generated_string_modifier ::= **"generated_string"** '(' *variable* ')'
statement

The *variable* argument gives the name of the variable that will be populated with the generated text. This variable must already exist, declared on the stack or referring a node of the current parse tree.

Statement modifier 'appended file'

This keyword stands just before an instruction or a compound statement that belongs to a *pattern script* exclusively. A new output file is opened for appending source code generation at the end of the file and replaces temporarily the current one during the execution of the statement. The statement is executed and, once the controlling sequence leaves the statement, the output file is closed properly and the precedent one takes its place.

The BNF representation of the `appended_file` statement modifier looks like:
appended_file_modifier ::= **"appended_file"** '(' *filename* ')'
statement

The token *filename* is an expression that is evaluated to give the name of the output file to append.

4.3 Common functions and procedures

All functions and procedures that are described below may be encountered in any kind of scripts : parsing, source code generation and file expanding, process driving, included script files.

Category *interpreter*	Function for running a CODEWORKER **script**
`autoexpand`	Expands a file on markups, following the directives self-contained in the file.
`executeString`	Executes a script given in a string.
`executeStringQuiet`	Interprets a string as a script and returns all traces intended to the console.
`expand`	Expands a file on markups, following the directives of a template-based script.
`extendExecutedScript`	Extend the current executed script dynamically with the content of the string.
`generate`	Generates a file, following the directives of a template-based script.
`generateString`	Generates a string, following the directives of a template-based script.
`parseAsBNF`	Parses a file with a BNF script.
`parseFree`	Parses a file with an imperative script.
`parseFreeQuiet`	Parses a file with an imperative script, reroute all console messages and returns them as a string.
`parseStringAsBNF`	Parses a string with a BNF script.
`traceEngine`	Displays the state of the interpreter.
`translate`	Performs a *source-to-source translation* or a *program transformation*.
`translateString`	Performs a *source-to-source translation* or a *program transformation* on strings.

Category *string*	Functions for handling strings
charAt	Returns the characters present at a given position of a string.
completeLeftSpaces	Completes a string with spaces to the left so that it reaches a given size.
completeRightSpaces	Completes a string with spaces to the right so that it reaches a given size.
composeAdaLikeString	Converts a sequence of characters to a Ada-like string without double quote delimiters.
composeCLikeString	Converts a sequence of characters to a C-like string without double quote delimiters.
composeHTMLLikeString	Converts a sequence of characters to an HTML-like text
composeSQLLikeString	Converts a sequence of characters to a SQL-like string without single quote delimiters.
coreString	Extracts the core of a string, leaving the beginning and the end.
countStringOccurences	How many occurences of a string to another.
cutString	Cuts a string at each separator encountered.
endString	Compares the end of the string.
endl	Returns an end-of-line, depending on the operating system.
equalsIgnoreCase	Compares two strings, ignoring the case.
executeString	Executes a script given in a string.
executeStringQuiet	Interprets a string as a script and returns all traces intended to the console.
findFirstChar	Returns the position of the first character amongst a set, encountered into a string.
findLastString	Returns the position of the last occurence of a string to another.
findNextString	Returns the next occurence of a string to another.
findString	Returns the first occurence of a string to another.
generateString	Generates a string, following the directives of a template-based script.
joinStrings	Joins a list of strings, adding a separator between them.
leftString	Returns the beginning of a string.
lengthString	Returns the length of a string.
midString	Returns a substring starting at a point for a given length.
parseStringAsBNF	Parses a string with a BNF script.
repeatString	Returns the concatenation of a string repeated a few times.
replaceString	Replaces a substring with another.
replaceTabulations	Replaces tabulations with spaces.
rightString	Returns the end of a string.
rsubString	Returns the left part of a string, ignoring last characters.
startString	Checks the beginning of a string.
subString	Returns a substring, ignoring the first characters.
toLowerString	Converts a string to lowercase.
toUpperString	Converts a string to uppercase.
trim	Eliminates heading and trailing whitespaces.
trimLeft	Eliminates the leading whitespaces.
trimRight	Eliminates the trailing whitespaces.
truncateAfterString	Special truncation of a string.
truncateBeforeString	Special truncation of a string.

Category *array*	**Functions handling arrays**
`findElement`	Checks the existence of an entry key in an array.
`findFirstSubstringIntoKeys`	Returns the first entry key of an array, containing a given string.
`findNextSubstringIntoKeys`	Returns the next entry key of an array, containing a given string.
`getArraySize`	Returns the number of items in an array.
`insertElementAt`	Inserts a new element to a list, at a given position.
`invertArray`	Inverts the order of items in an array.
`isEmpty`	Checks whether a node has items or not.
`removeAllElements`	Removes all items of the array.
`removeElement`	Removes an item, given its entry key.
`removeFirstElement`	Removes the first item of the array.
`removeLastElement`	Removes the last item of the array.

Category *node*	**Functions handling a node**
`clearVariable`	Removes the subtree and assigns an empty value.
`equalTrees`	Compares two subtrees.
`existVariable`	Checks the existence of a node.
`getVariableAttributes`	Extract all attribute names of a tree node.
`removeRecursive`	Removes a given attribute from the subtree.
`removeVariable`	Removes a given variable.
`slideNodeContent`	Moves the subtree elsewhere on a branch.
`sortArray`	Sort an array, considering the entry keys.

Category *iterator*	**Functions handling an iterator**
`createIterator`	Creates an iterator pointing to the beginning of a list.
`createReverseIterator`	Creates a reverse iterator pointing to the end of a list.
`duplicateIterator`	Duplicates an iterator.
`first`	Returns `true` if the iterator points to the first item.
`index`	Returns the position of an item in a list.
`key`	Returns the entry key of the item pointed to by the iterator.
`last`	Returns `true` if the iterator points to the last item.
`next`	Move an iterator to the next item of a list.
`prec`	Move an iterator to the precedent item of a list.

Category *file*	Functions handling files
appendFile	Writes the content of a string to the end of a file
canonizePath	Builds an absolute path, starting to the current directory.
changeFileTime	Changes the access and modification times of a file.
chmod	Changes the permissions of a file.
copyFile	Copies a file.
copyGenerableFile	Copies a file with protected areas or expandable markups, only if the hand-code differs between source and destination.
copySmartFile	Copies a file only if the destination differs.
createVirtualFile	Creates a transient file in memory.
createVirtualTemporaryFile	Creates a transient file in memory, CODEWORKER choosing its name.
deleteFile	Deletes a file on the disk.
deleteVirtualFile	Deletes a transient file from memory.
existFile	Checks the existence of a file.
existVirtualFile	Checks the existence of a transient file, created in memory.
exploreDirectory	Browses all files of a directory, recursively or not.
fileCreation	Returns the creation date of a file.
fileLastAccess	Returns the last access date of a file.
fileLastModification	Returns the last modification date of a file.
fileLines	Returns the number of lines in a file.
fileMode	Returns the permissions of a file.
fileSize	Returns the size of a file.
getGenerationHeader	Returns the comment to put into the header of generated files.
getShortFilename	Returns the short name of a file
indentFile	Indents a file, depending on the target language.
loadBinaryFile	Loads a binary file and stores each byte in a hexadecimal representation of 2
loadFile	Returns the content of a file or raises an error if not found.
loadVirtualFile	Returns the content of a transient file or raises an error if not found.
pathFromPackage	Converts a package path to a directory path.
relativePath	Returns the relative path, which allows going from a path to another.
resolveFilePath	Gives the location of a file with no ambiguity.
saveBinaryToFile	Saves binary data to a file.
saveToFile	Saves the content of a string to a file
scanDirectories	Explores a directory, filtering filenames.
scanFiles	Returns a flat list of all filenames matching with a filter.

Category *directory*	Functions handling directories
changeDirectory	Changes the current directory (`chdir()` in C).
copySmartDirectory	Copies files of a directory recursively only when destination files differ from source files.
createDirectory	Creates a new directory.
existDirectory	Check the existence of a directory.
exploreDirectory	Browses all files of a directory, recursively or not.
getCurrentDirectory	Returns the current directory (`getcwd()` in C).
removeDirectory	Removes a directory from the disk.
scanDirectories	Explores a directory, filtering filenames.
scanFiles	Returns a flat list of all filenames matching with a filter.

Category *URL*	Functions working on URL transfers (HTTP,...)
`decodeURL`	Decodes an HTTP URL.
`encodeURL`	Encodes an URL to HTTP.
`getHTTPRequest`	Sends an HTTP's GET request.
`postHTTPRequest`	Sends an HTTP's POST request.
`sendHTTPRequest`	Sends an HTTP request.

Category *datetime*	Functions handling date-time
`addToDate`	Change a date by shifting its internal fields days/months/years or time.
`compareDate`	Compares two dates.
`completeDate`	Extends an incomplete date with *today* characteristics.
`fileCreation`	Returns the creation date of a file.
`fileLastAccess`	Returns the last access date of a file.
`fileLastModification`	Returns the last modification date of a file.
`formatDate`	Changes the format of a date.
`getLastDelay`	Returns the time consumed to execute a statement.
`getNow`	Returns the current date-time.
`setNow`	Fixes the current date-time.

Category *numeric*	Functions handling numbers
`add`	Equivalent admitted writing is `$a + b$`.
`ceil`	Returns the smallest integer greater that or equal to a number
`decrement`	Equivalent admitted writing is `set a = $a - 1$;`.
`div`	Equivalent admitted writing is `$a / b$`.
`equal`	Equivalent admitted writing is `$a == b$`.
`exp`	Returns the exponential of a value.
`floor`	Returns the largest integer less that or equal to a number
`increment`	Equivalent admitted writing is `set a = $a + 1$;`.
`inf`	Equivalent admitted writing is `$a < b$`.
`isNegative`	Equivalent admitted writing is `$a < 0$`.
`isPositive`	Equivalent admitted writing is `$a > 0$`.
`log`	Returns the Neperian logarithm.
`mod`	Equivalent admitted writing is `$a %b$`.
`mult`	Equivalent admitted writing is `$a * b$`.
`pow`	Raises a number to the power of another.
`sqrt`	Calculates the square root.
`sub`	Equivalent admitted writing is `$a - b$`.
`sup`	Equivalent admitted writing is `$a > b$`.

Chapter 4. The scripting language

Category *standard*	Classical functions of any standard library
`UUID`	Generates an UUID.
`error`	Raises an error message
`inputKey`	If any, returns the last key pressed on the standard input.
`inputLine`	Wait for the standard input to the console.
`isIdentifier`	Checks whether a string is a C-like identifier or not.
`isNumeric`	Checks whether a string is a floating-point number or not.
`randomInteger`	Generates a pseudorandom number.
`randomSeed`	Changes the seed of the pseudorandom generator.
`traceLine`	Displays a message to the console, adding a carriage return.
`traceObject`	Displays the content of a node to the console.
`traceStack`	Displays the stack to the console.
`traceText`	Displays a message to the console.

Category *conversion*	Type conversion
`byteToChar`	Converts a byte (hexadecimal representation of 2 digits) to a character.
`bytesToLong`	Converts a 4-bytes sequence to an unsigned long integer in its decimal representation.
`bytesToShort`	Converts a 2-bytes sequence to an unsigned short integer in its decimal representation.
`charToByte`	Converts a character to a byte (hexadecimal representation of 2 digits).
`charToInt`	Converts a character to the integer value of the corresponding ASCII.
`hexaToDecimal`	Converts an hexadecimal representation to an integer.
`hostToNetworkLong`	Converts a 4-bytes representation of a long integer to the network bytes order.
`hostToNetworkShort`	Converts a 2-bytes representation of a short integer to the network bytes order.
`longToBytes`	Converts an unsigned long integer in decimal base to its 4-bytes representation.
`networkLongToHost`	Converts a 4-bytes representation of a long integer to the host bytes order.
`networkShortToHost`	Converts a 2-bytes representation of a short integer to the host bytes order.
`octalToDecimal`	Converts an octal representation to a decimal integer.
`shortToBytes`	Converts an unsigned short integer in decimal base to its 2-bytes representation.

Category *system*	Functions relative to the operating system
`computeMD5`	Computes the MD5 of a string.
`environTable`	Equivalent of `environ()` in C
`existEnv`	Checks the existence of an environment variable.
`getEnv`	Returns an environment variable, or raises an error if not exist.
`openLogFile`	Opens a log file for logging every console trace.
`putEnv`	Puts a value to an environment variable.
`sleep`	Suspends the execution for `millis` milliseconds.
`system`	Equivalent to the C function `system()`.

Category *command*	Relative to the command line
`compileToCpp`	Translates a script to C++.
`getIncludePath`	Returns the include path passed via the option `-I`.
`getProperty`	Returns the value of a property passed via the option `-D`.
`getVersion`	Returns the version of the interpreter.
`getWorkingPath`	Returns the output directory passed via option `-path`.
`setIncludePath`	Changes the option `-I` while running.
`setProperty`	Adds/changes a property (option `-D`) while running.
`setVersion`	Gives the version of scripts currently interpreted by *CodeWorker*.
`setWorkingPath`	Does the job of the option `-path`.

Category *generation*	Functions relative to generation
`addGenerationTagsHandler`	Adds your own CodeWorker's tags handler
`autoexpand`	Expands a file on markups, following the directives self-contained in the fil
`expand`	Expands a file on markups, following the directives of a template-based scr
`extractGenerationHeader`	Gives the generation header of a generated file, if any.
`generate`	Generates a file, following the directives of a template-based script.
`generateString`	Generates a string, following the directives of a template-based script.
`getCommentBegin`	Returns the current format of a comment's beginning.
`getCommentEnd`	Returns the current format of a comment's end.
`getGenerationHeader`	Returns the comment to put into the header of generated files.
`getTextMode`	Returns the text mode amongst **"DOS"**, **"UNIX"** and **"BINARY"**.
`getWriteMode`	Returns how text is written during a generation (insert/overwrite).
`listAllGeneratedFiles`	Gives the list of all generated files.
`removeGenerationTagsHandler`	Removes a custom generation tags handler
`selectGenerationTagsHandler`	Selects your own CodeWorker's tags handler for processing generation task
`setCommentBegin`	Changes what a beginning of comment looks like, perhaps before expandin
`setCommentEnd`	Changes what an end of comment looks like, perhaps before expanding a fi
`setGenerationHeader`	Specifies a comment to put at the beginning of every generated file.
`setTextMode`	**"DOS"**, **"UNIX"** or **"BINARY"**
`setWriteMode`	Selects how to write text during a generation (insert/overwrite).
`translate`	Performs a *source-to-source translation* or a *program transformation*.
`translateString`	Performs a *source-to-source translation* or a *program transformation* on str

Category *parsing*	Functions relative to scanning/parsing
`parseAsBNF`	Parses a file with a BNF script.
`parseFree`	Parses a file with an imperative script.
`parseFreeQuiet`	Parses a file with an imperative script, reroute all console messages and returns them as a string.
`parseStringAsBNF`	Parses a string with a BNF script.
`translate`	Performs a *source-to-source translation* or a *program transformation*.
`translateString`	Performs a *source-to-source translation* or a *program transformation* on strings.

Category *socket*	Socket operations
`acceptSocket`	Listens for a client connection and accepts it.
`closeSocket`	Closes a socket descriptor.
`createINETClientSocket`	Creates a stream socket connected to the specified port and IP address.
`createINETServerSocket`	Creates a server stream socket bound to a specified port.
`receiveBinaryFromSocket`	Reads binary data from the socket, knowing the size.
`receiveFromSocket`	Reads text or binary data from a socket.
`receiveTextFromSocket`	Reads text from a socket, knowing the size.
`sendBinaryToSocket`	Writes binary data to a socket.
`sendTextToSocket`	Writes text to a socket.

Category *unknown*	Various types of function
`loadProject`	Loads a parse tree previously saved thanks to **saveProject**().
`not`	The boolean negation, equivalent to `!a`.
`produceHTML`	
`saveProject`	Saves a parse tree to XML or to a particular text format.
`saveProjectTypes`	Factorizes nodes of the projects to distinguish implicit types for node and saves it to XML.

4.3.1 acceptSocket

- function **acceptSocket(**`serverSocket` **:** *int***) :** *int*

Parameter	Type	Description
`serverSocket`	*int*	a server socket previously created via `createINETServerSocket()`

This function blocks until a client connection arrives, and returns the corresponding socket descriptor.

Once a connection has been established, use directly the *send*/*receive* functions or `attachInputToSocket()`/`attachOutputToSocket` for reading/writing to the socket via a *BNF-parsing*/*template-based* script.

See also:

`createINETClientSocket` 4.3.36, `createINETServerSocket` 4.3.37, `attachInputToSocket` 4.5, `detachInputFromSocket` 4.5.3, `attachOutputToSocket` 4.6.3, `detachOutputFromSocket` 4.6.7, `receiveBinaryFromSocket` 4.3.153, `receiveFromSocket` 4.3.154, `receiveTextFromSocket` 4.3.155, `sendTextToSocket` 4.3.180, `sendBinaryToSocket` 4.3.178, `closeSocket` 4.3.18, `flushOutputToSocket` 4.6.10

4.3.2 add

- function **add(**`left` **:** *double***,** `right` **:** *double***) :** *double*

Parameter	Type	Description
`left`	*double*	left arithmetic member
`right`	*double*	right arithmetic member

Returns the result of arithmetic addition `left` **+** `right`. Members are converted from strings to numbers, supposed being worth 0 if a parsing error occurs; then the addition is processed, and the result is converted to a string, skipping fractional part if all digits after the dot are *0*.

Remember that the symbol **'+'** means the concatenation of text. Using this operator instead of function `add` will concatenate digits! However, it exists an escape mode that allows writing arithmetic expressions between **'$'** symbols, as formula under *LaTeX*. So, `$left + right$` is equivalent to `add(left, right)`.

Example:

```
local a = 3.2;
traceLine(a + " + 4.5 = " + add(a, "4.5"));
traceLine(a + " + 2.8 = " + add(a, 2.8) + " <- integer value");
```

Output:

```
3.2 + 4.5 = 7.7
3.2 + 2.8 = 6 <- integer value
```

See also:

`sub` 4.3.197, `mult` 4.3.134, `div` 4.3.47, `exp` 4.3.64, `log` 4.3.130, `mod` 4.3.133, `pow` 4.3.147

4.3.3 addGenerationTagsHandler

- function **addGenerationTagsHandler**(key : *string*, reader : *script*, writer : *script*) : *bool*

Parameter	Type	Description
key	*string*	designates the handler
reader	*script<BNF>*	extended-BNF script of the reader
writer	*script<pattern>*	template-based script of the writer

Adds a new generation tags handler, designated by key.

Returns true if key isn't reserved yet for another generation tags handler.

See also:

removeGenerationTagsHandler   4.3.161,   selectGenerationTagsHandler
4.3.177

4.3.4 addToDate

- function **addToDate**(date : *string*, format : *string*, shifting : *string*) : *string*

Parameter	Type	Description
date	*string*	the date to change
format	*string*	the format to apply on the reading of the shifting argument
shifting	*string*	the offset values to apply on the date, whose meanings are known by the offset argument

Change a date by applying offset values on its internal representation. The internal representation holds the year / month / day and hour / minute / second and millisecond fields. You choose what fields to shift, giving a date format as the first argument, and an offset value for each fields seen in the format as the second argument.

The field types have the same syntax as in the function formatDate, except that the field values might be negative.
For instance, if the field type is "%m", the month must occupy 2 digits maximum for a positive offset, and 3 characters for a negative offset, the first one being the sign.

The offsets are applied in the order they are read, from the left-hand side to the right.

The function returns the value of the date after applying the shift.

Example:

```
traceLine("Substract 2 months and add 20 hours to the current
date-time:");
local newDate = addToDate(getNow(), "%m,%H", "-2,20");
traceLine("one manner:  " + getNow() + " -> " + newDate);
newDate = addToDate(getNow(), "%m%H", "-0220");
traceLine("another manner:  " + getNow() + " -> " + newDate);
```

Output:

```
Substract 2 months and add 20 hours to the current date-time:
one manner:  28jul2008 20:42:00.500 -> 29may2008 16:42:00.500
```

another manner: `28jul2008 20:42:00.500 -> 29may2008 16:42:00.500`

See also:

`formatDate` 4.3.84, `compareDate` 4.3.19, `completeDate` 4.3.21, `getLastDelay` 4.3.95, `getNow` 4.3.96, `setNow` 4.3.185

4.3.5 appendFile

- procedure **appendFile(**`filename` **:** *string***,** `content` **:** *string***)**

Parameter	Type	Description
`filename`	*string*	name of the file to append
`content`	*string*	sequence of characters to write at the end of the file

Writes the text `content` at the end of the file `filename`.

If the file doesn't exist, the function creates it.

See also:

`copyFile` 4.3.29, `changeFileTime` 4.3.12, `chmod` 4.3.16, `copyGenerableFile` 4.3.30, `copySmartFile` 4.3.32, `deleteFile` 4.3.45, `existFile` 4.3.61, `fileCreation` 4.3.69, `fileLastAccess` 4.3.70, `fileLastModification` 4.3.71, `fileLines` 4.3.72, `fileMode` 4.3.73, `fileSize` 4.3.74, `loadBinaryFile` 4.3.126, `loadFile` 4.3.127, `saveBinaryToFile` 4.3.171, `saveToFile` 4.3.174, `scanFiles` 4.3.176

4.3.6 autoexpand

- procedure **autoexpand(**`outputFileName` **:** *string***,** `this` **:** *treeref***)**

Parameter	Type	Description
`outputFileName`	*string*	the existing file to expand
`this`	*treeref*	the current node that will be accessed with *this* variable

Expands an existing file whose name is passed to the argument `outputFileName`, executing *template-based* scripts located at each markup. The file contains its own scripts for expanding code.

Expanding a file consists of generating code into marked out areas only, the rest of the file remaining the same. The markup is put into a comment, knowing that the syntax of the comment must conform to the type of the expanded file `outputFileName`. So, an HTML file expects **<!- -** and **- ->**, a JAVA file is waiting for **//** and **"\n"**, ... The markup is announced by **##markup##** followed by a string that represents the *markup key*. Don't forget to configure correctly the syntax of comment boundaries with procedures `setCommentBegin()` (see 4.3.181) and `setCommentEnd()` (see 4.3.182).

When the procedure is called, CODEWORKER jumps from a markup to another. To handle a markup, it checks whether text was already generated, put between tags **##begin##"*markup-key*"** and **##end##"*markup-key*"**, added automatically the first time an expansion is required, to demarquate the portion of code that doesn't belong to the user. Then, it extracts all protected areas, if any, and it generates code at the position of the markup, adding *begin/end* tags seen before.

The interpreter reclaims the tags `##script##` just after the markup. It extracts the embedded text, considered as a `template-based` script, eventually put between comments, and the interpreter executes this embedded script.

Note that some data might be put between tags `##data##`, accessible in the *template-based* script via the function `getMarkupValue()` (see 4.6.14). This block of custom data comes after the `##script##` tag, if present.

Be careful not to confuse this prodedure with `generate()` that doesn't care about markups and that overwrites the output file completely, except protected areas of course.

See also:

`expand` 4.3.65, `generate` 4.3.85, `generateString` 4.3.86, `translate` 4.3.208, `parseAsBNF` 4.3.141, `parseFree` 4.3.142, `parseFreeQuiet` 4.3.143, `parseStringAsBNF` 4.3.144, `translateString` 4.3.209

4.3.7   bytesToLong

- function **bytesToLong**(`bytes` : *string*) : *ulong*

Parameter	Type	Description
`bytes`	*string*	a 4-bytes representation of an unsigned long integer (host bytes order)

Converts a 4-bytes representation of an unsigned long integer to its decimal representation. Bytes are ordered in the host order (memory storage).

If the argument `bytes` is malformed, the function raises an error.

Example:

`traceLine("bytesToLong('FFFFFFFF') = '" + bytesToLong("FFFFFFFF") + "'");`

Output:

`bytesToLong('FFFFFFFF') = '4294967295'`

See also:

`byteToChar` 4.3.8, `bytesToShort` 4.3.7, `charToByte` 4.3.14, `charToInt` 4.3.15, `hexaToDecimal` 4.3.104, `longToBytes` 4.3.131, `octalToDecimal` 4.3.139, `shortToBytes` 4.3.191

4.3.8   bytesToShort

- function **bytesToShort**(`bytes` : *string*) : *ushort*

Parameter	Type	Description
`bytes`	*string*	a 2-bytes representation of an unsigned short integer (host bytes order)

Converts a 2-bytes representation of an unsigned short integer to its decimal representation. Bytes are ordered in the host order (memory storage).

If the argument `bytes` is malformed, the function raises an error.

Example:

```
traceLine("bytesToShort('FFFF') = '" + bytesToShort("FFFF") +
"'");
```

Output:

```
bytesToShort('FFFF') = '65535'
```

See also:

```
byteToChar 4.3.8, bytesToLong 4.3.6, charToByte 4.3.14, charToInt 4.3.15,
hexaToDecimal 4.3.104, longToBytes 4.3.131, octalToDecimal 4.3.139,
shortToBytes 4.3.191
```

4.3.9 byteToChar

- function **byteToChar**(`byte` : *string*) : *string*

Parameter	Type	Description
`byte`	*string*	an hexadecimal number of 2 digits exactly

Converts a byte to a character. A byte is considered as an hexadecimal number of 2 digits exactly.

If the argument `byte` doesn't contain an hexadecimal number of 2 digits, an error is raised. If `byte` is worth '00', the function returns an empty string.

Example:

```
traceLine("byteToChar('20') = '" + byteToChar("20") + "'");
traceLine("byteToChar('61') = '" + byteToChar("61") + "'");
```

Output:

```
byteToChar('20') = ' '
byteToChar('61') = 'a'
```

See also:

```
bytesToLong 4.3.6, bytesToShort 4.3.7, charToByte 4.3.14, charToInt
4.3.15, hexaToDecimal 4.3.104, longToBytes 4.3.131, octalToDecimal 4.3.139,
shortToBytes 4.3.191
```

4.3.10 canonizePath

- function **canonizePath**(`path` : *string*) : *string*

Parameter	Type	Description
`path`	*string*	the path to canonize

Returns the path passed to the argument `path` after having canonized it.

To canonize a path means that:

- '..' and '.' directories are processed,
- backslashes are changed to forward slashes,
- if the `path` is relative, it is converted to a full path, starting at the current directory.

Example:

```
traceLine("current directory = '" + getCurrentDirectory() +
"'");
local sPath = "WebSite/downloads/CodeWorker.zip";
traceLine(" path = '" + sPath + "'");
traceLine(" result = '" + canonizePath(sPath) + "'");
local sCurrentDirectory = getCurrentDirectory();
changeDirectory(sCurrentDirectory + "Documentation");
traceLine("current directory = '" + getCurrentDirectory() +
"'");
set sPath = "../Scripts/Tutorial/GettingStarted/tiny.html";
traceLine(" path = '" + sPath + "'");
traceLine(" result = '" + canonizePath(sPath) + "'");
changeDirectory(sCurrentDirectory);
traceLine("current directory = '" + getCurrentDirectory() +
"'");
set sPath = ".";
traceLine(" path = '" + sPath + "'");
traceLine(" result = '" + canonizePath(sPath) + "'");
```

Output:

```
current directory = 'E:/projects/generator/'
    path = 'WebSite/downloads/CodeWorker.zip'
    result = 'e:/projects/generator/WebSite/downloads/CodeWorker.zip'
current directory = 'E:/projects/generator/Documentation/'
    path = '../Scripts/Tutorial/GettingStarted/tiny.html'
    result = 'e:/projects/generator/Scripts/Tutorial/GettingStarted/tiny.html
current directory = 'E:/projects/generator/'
    path = '.'
    result = 'e:/projects/generator'
```

See also:

`changeDirectory` 4.3.11, `copySmartDirectory` 4.3.31, `exploreDirectory` 4.3.66,
`getCurrentDirectory` 4.3.90, `relativePath` 4.3.156, `removeDirectory` 4.3.158,
`resolveFilePath` 4.3.168, `scanDirectories` 4.3.175, `existDirectory` 4.3.59

4.3.11 ceil

- function **ceil**(number **:** *double*) **:** *int*

Parameter	Type	Description
number	*double*	the floating-point number to ceil

Returns the smallest integer that is greater than or equal to `number`. If `number` isn't a number,
the function returns *0*.

Example:

```
traceLine("ceil(5.369e+1) = " + ceil(5.369e1));
```

Output:

```
ceil(5.369e+1) = 54
```

decrement 4.3.44, increment 4.3.107, floor 4.3.83

4.3.12   changeDirectory

- function **changeDirectory**(path : *string*) : *bool*

Parameter	Type	Description
path	*string*	path name of the directory

The function changes the current directory of CODEWORKER to the directory specified by the path argument. The parameter must refer to an existing directory.

Example:

```
traceLine("current directory = '" + getCurrentDirectory() +
"'");
local sOldDirectory = getCurrentDirectory();
local sNewDirectory = sOldDirectory + "Documentation";
traceLine("call to changeDirectory('" + sNewDirectory + "')");
changeDirectory(sNewDirectory);
traceLine("new current directory = '" + getCurrentDirectory() +
"'");
changeDirectory(sOldDirectory);
```

Output:

```
current directory = 'E:/projects/generator/'
call to changeDirectory('E:/projects/generator/Documentation')
new current directory = 'E:/projects/generator/Documentation/'
```

See also:

canonizePath 4.3.9, copySmartDirectory 4.3.31, exploreDirectory 4.3.66, getCurrentDirectory 4.3.90, relativePath 4.3.156, removeDirectory 4.3.158, resolveFilePath 4.3.168, scanDirectories 4.3.175, existDirectory 4.3.59

4.3.13   changeFileTime

- function **changeFileTime**(filename : *string*, accessTime : *string*, modificationTime : *string*) : *int*

Parameter	Type	Description
filename	*string*	name of the file to set
accessTime	*string*	date-time of the last access
modificationTime	*string*	date-time of the last modification

The function changes the access and modification times of the file filename. The user ID of the process must be the owner of the file, or the process must have appropriate privileges.

In case of failure, the function returns a negative integer:

- **-1**: unknown error that shouldn't appear,

- **-2**: permission denied,
- **-3**: too many files have been opened,
- **-4**: file not found,
- **-5**: invalid *times* argument,

Example:

```
local oldAccessTime = fileLastAccess("readme.txt");
local oldModifTime = fileLastModification("readme.txt");
traceLine("old modification time of 'readme.txt' = '" +
oldModifTime + "'");
if $changeFileTime("readme.txt", getNow(), getNow()) < 0$
    error("'changeFileTime()' has failed!");
local newModifTime = fileLastModification("readme.txt");
traceLine("new modification time of 'readme.txt' = '" +
newModifTime + "'");
// put the same times as before calling the example:
if $changeFileTime("readme.txt", oldAccessTime, oldModifTime) <
0$
    error("'changeFileTime()' has failed!");
```

Output:

```
old modification time of 'readme.txt' = '02may2008 04:51:36'
new modification time of 'readme.txt' = '28jul2008 20:42:00'
```

See also:

`copyFile` 4.3.29, `appendFile` 4.3.4, `chmod` 4.3.16, `copyGenerableFile` 4.3.30, `copySmartFile` 4.3.32, `deleteFile` 4.3.45, `existFile` 4.3.61, `fileCreation` 4.3.69, `fileLastAccess` 4.3.70, `fileLastModification` 4.3.71, `fileLines` 4.3.72, `fileMode` 4.3.73, `fileSize` 4.3.74, `loadBinaryFile` 4.3.126, `loadFile` 4.3.127, `saveBinaryToFile` 4.3.171, `saveToFile` 4.3.174, `scanFiles` 4.3.176

4.3.14   charAt

- function **charAt**(`text` **:** *string***,** `index` **:** *int***) :** *string*

Parameter	Type	Description
`text`	*string*	a sequence of characters
`index`	*int*	the index of the character to extract from `text`

Returns the character at the specified `index`. An index ranges from *0* to **lengthString**(`text`) **- 1**. The first character of the sequence is at index *0*, the next at index *1*, and so on. If the `index` argument is out of bounds (negative or not less than the length of `text`), it returns an empty string.

Example:

```
local sText = "I have but one lamp by which my feet are guided,
and that is the lamp of experience.  (P. Henry)";
traceLine("charAt('" + sText + "', 2) = '" + charAt(sText, 2) +
"' <- index = 2 gives the third character of the string");
```

Output:

```
charAt('I have but one lamp by which my feet are guided, and
that is the lamp of experience.  (P. Henry)', 2) = 'h' <- index
= 2 gives the third character of the string
```

See also:

`coreString` 4.3.33, `cutString` 4.3.42, `joinStrings` 4.3.120, `leftString` 4.3.123, `lengthString` 4.3.124, `midString` 4.3.132, `rightString` 4.3.169, `rsubString` 4.3.170, `subString` 4.3.198

4.3.15 charToByte

- function **charToByte(**`char` **:** *string***) :** *string*

Parameter	Type	Description
`char`	*string*	a character

Converts a character to its hexadecimal representation, taking 2 digits, even if less than 0x10.

If the argument `char` is empty, the function returns '00'. If it contains more than one character, an error is raised.

Example:

```
traceLine("charToByte('A') = '" + charToByte("A") + "'");
traceLine("charToByte('\\n') = '" + charToByte("\n") + "'");
```

Output:

```
charToByte('A') = '41'
charToByte('\n') = '0A'
```

See also:

`byteToChar` 4.3.8, `bytesToLong` 4.3.6, `bytesToShort` 4.3.7, `charToInt` 4.3.15, `hexaToDecimal` 4.3.104, `longToBytes` 4.3.131, `octalToDecimal` 4.3.139, `shortToBytes` 4.3.191

4.3.16 charToInt

- function **charToInt(**`char` **:** *string***) :** *int*

Parameter	Type	Description
`char`	*string*	a string containing just one char

Returns the conversion of `char` as an unsigned integer, corresponding to its ASCII form generally. If `char` doesn't contain just one char, it returns an empty string.

Example:

```
traceLine("charToInt('A') = " + charToInt("A") + " <- ASCII code
of 'A'");
```

Output:

```
charToInt('A') = 65 <- ASCII code of 'A'
```

See also:

byteToChar 4.3.8, bytesToLong 4.3.6, bytesToShort 4.3.7, charToByte 4.3.14, hexaToDecimal 4.3.104, longToBytes 4.3.131, octalToDecimal 4.3.139, shortToBytes 4.3.191

4.3.17   chmod

- function **chmod(**filename **:** *string,* mode **:** *string***) :** *bool*

Parameter	Type	Description
filename	*string*	file to which change the permission setting
mode	*string*	permission setting as a concatenation of 'R' and/or 'W' and/or 'X'

The chmod function changes the permission setting of the file specified by filename. The permission setting controls *read* and *write* and *execute* access to the file. The argument mode holds the permission setting of the file as a concatenation of chars amongst the following:

- **'R'** for **r**eading permitted,
- **'W'** for **w**riting permitted,
- **'X'** for e**x**ecuting permitted (ignored on *Windows* platform),

The function fails when the file given by the argument filename is not found, and an error is thrown when the argument mode contains an unexpected character.

Example:

```
local bSuccess = chmod("Documentation/CodeWorker.tex", "RW");
if !bSuccess error("file 'Documentation/CodeWorker.tex' not
found!");
traceLine("R + W permitted on file 'Documentation/CodeWorker.tex'");
```

Output:

```
R + W permitted on file 'Documentation/CodeWorker.tex'
```

See also:

copyFile 4.3.29, appendFile 4.3.4, changeFileTime 4.3.12, copyGenerableFile 4.3.30, copySmartFile 4.3.32, deleteFile 4.3.45, existFile 4.3.61, fileCreation 4.3.69, fileLastAccess 4.3.70, fileLastModification 4.3.71, fileLines 4.3.72, fileMode 4.3.73, fileSize 4.3.74, loadBinaryFile 4.3.126, loadFile 4.3.127, saveBinaryToFile 4.3.171, saveToFile 4.3.174, scanFiles 4.3.176

4.3.18   clearVariable

- procedure **clearVariable(**node **:** *treeref* **)**

Parameter	Type	Description
node	*treeref*	the node to clear

All attributes of the argument node are deleted, its array of nodes is cleared and its value becomes an empty string. If the node was referring to another node, the link is cleared.

Please note that the node isn't removed; see `removeVariable()` for that.

Example:

```
local myNode = "the value";
insert myNode.a1 = "attribute 1";
insert myNode.a2 = "attribute 2";
insert myNode.array["1"] = "node 1";
insert myNode.array["2"] = "node 2";
traceObject(myNode);
traceLine("- the variable 'myNode' is cleared:");
clearVariable(myNode);
traceObject(myNode);
```

Output:

```
Tracing variable 'myNode':
    "the value"
    a1 = "attribute 1"
    a2 = "attribute 2"
    array
    array["1", "2"]
End of variable's trace 'myNode'.
- the variable 'myNode' is cleared:
Tracing variable 'myNode':
End of variable's trace 'myNode'.
```

Deprecated form: `clearNode` has disappeared since version *3.8.7*

See also:

`existVariable` 4.3.62, `findFirstSubstringIntoKeys` 4.3.77, `findElement` 4.3.75, `findNextSubstringIntoKeys` 4.3.80, `getArraySize` 4.3.87, `getVariableAttributes` 4.3.100, `invertArray` 4.3.114, `isEmpty` 4.3.115, `removeVariable` 4.3.164

4.3.19 closeSocket

- procedure **closeSocket(**`socket` **:** *int)*

Parameter	Type	Description
`socket`	*int*	a *client/server* socket descriptor

This procedure closes the socket descriptor specified to the argument `socket`.

See also:

`createINETClientSocket` 4.3.36, `createINETServerSocket` 4.3.37, `acceptSocket` 4.3, `attachInputToSocket` 4.5, `detachInputFromSocket` 4.5.3, `attachOutputToSocket` 4.6.3, `detachOutputFromSocket` 4.6.7, `receiveBinaryFromSocket` 4.3.153, `receiveFromSocket` 4.3.154, `receiveTextFromSocket` 4.3.155, `sendTextToSocket` 4.3.180, `sendBinaryToSocket` 4.3.178, `flushOutputToSocket` 4.6.10

4.3.20 compareDate

- function **compareDate(**date1 **:** *string***,** date2 **:** *string***) :** *int*

Parameter	Type	Description
date1	*string*	a date that conforms to the following format: `"%d%b%Y %H:%M:%S.%L"`
date2	*string*	second date to compare to date1

The function returns:

- a **negative value** when *date1 < date2*,
- **zero** when *date1 is equal to date2*,
- a **positive value** when *date1 > date2*.

If an argument doesn't conform to the expected syntax for a date (meaning that it must match with `"%d%b%Y %H:%M:%S.%L"`), an error is raised.

Example:

```
local date1 = "19jan2003 20:12:00.000";
local date2 = "28dec2012 07:30:00.000";
local now = getNow();
traceLine("getNow() = '" + now + "'");
traceLine("compareDate('" + date1 + "', getNow()) = " +
compareDate(date1, now));
traceLine("compareDate('" + date2 + "', getNow()) = " +
compareDate(date2, now));
```

Output:

```
getNow() = '28jul2008 20:42:00.500'
compareDate('19jan2003 20:12:00.000', getNow()) = -1
compareDate('28dec2012 07:30:00.000', getNow()) = 1
```

See also:

formatDate 4.3.84, addToDate 4.3.3, completeDate 4.3.21, getLastDelay 4.3.95, getNow 4.3.96, setNow 4.3.185

4.3.21 compileToCpp

- procedure **compileToCpp(**scriptFileName **:** *string***,** projectDirectory **:** *string***,** CodeWorkerDirectory **:** *string***)**

Parameter	Type	Description
scriptFileName	*string*	the name of a script file to compile to C++
projectDirectory	*string*	the location where to put on the disk the scripts compiled to C++
CodeWorkerDirectory	*string*	the root path of CODEWORKER either in development or distributed state

Compiles the leader script file called `scriptFileName` and all scripts that might be reclaimed during the execution. The corresponding C++ files are stored into `projectDirectory` with the makefile (a Visual C++'s *DSP* file). The path to libraries and the origin of some important include files is determined thanks to the path to CODEWORKER put into `CodeWorkerDirectory`.

The script file cannot be a *pattern script* or a *parsing* script.

If an error occurs, an error message is raised.

Example:

```
local sScriptFile = "GettingStarted/LeaderScript6.cws";
local sDirectory = getWorkingPath() +
            "Scripts/Tutorial/GettingStarted/bin";
removeDirectory(sDirectory);
compileToCpp(sScriptFile, sDirectory, ".");
local theFiles;
if !scanFiles(theFiles, sDirectory, "", true) error("should have
worked!");
traceLine("generated files:");
foreach i in sorted theFiles traceLine(" " + i);
```

Output:

```
generated files:
    e:\Projects\generator/Scripts/Tutorial/GettingStarted/bin/CGExternalHandl
    e:\Projects\generator/Scripts/Tutorial/GettingStarted/bin/CGRuntime.h
    e:\Projects\generator/Scripts/Tutorial/GettingStarted/bin/CppObjectBody_c
    e:\Projects\generator/Scripts/Tutorial/GettingStarted/bin/CppObjectBody_c
    e:\Projects\generator/Scripts/Tutorial/GettingStarted/bin/CppObjectHeader
    e:\Projects\generator/Scripts/Tutorial/GettingStarted/bin/CppObjectHeader
    e:\Projects\generator/Scripts/Tutorial/GettingStarted/bin/CppParsingTree.
    e:\Projects\generator/Scripts/Tutorial/GettingStarted/bin/DynPackage.h
    e:\Projects\generator/Scripts/Tutorial/GettingStarted/bin/HTML2LaTeX_cwp.
    e:\Projects\generator/Scripts/Tutorial/GettingStarted/bin/HTML2LaTeX_cwp.
    e:\Projects\generator/Scripts/Tutorial/GettingStarted/bin/HTMLDocumentati
    e:\Projects\generator/Scripts/Tutorial/GettingStarted/bin/HTMLDocumentati
    e:\Projects\generator/Scripts/Tutorial/GettingStarted/bin/JAVAObject_cwt.
    e:\Projects\generator/Scripts/Tutorial/GettingStarted/bin/JAVAObject_cwt.
    e:\Projects\generator/Scripts/Tutorial/GettingStarted/bin/LeaderScript6.c
    e:\Projects\generator/Scripts/Tutorial/GettingStarted/bin/LeaderScript6_c
    e:\Projects\generator/Scripts/Tutorial/GettingStarted/bin/LeaderScript6_c
    e:\Projects\generator/Scripts/Tutorial/GettingStarted/bin/Makefile
    e:\Projects\generator/Scripts/Tutorial/GettingStarted/bin/SimpleML-parsin
    e:\Projects\generator/Scripts/Tutorial/GettingStarted/bin/SimpleML-parsin
    e:\Projects\generator/Scripts/Tutorial/GettingStarted/bin/UtlException.h
```

4.3.22 completeDate

- function **completeDate**(`date` **:** *string***,** `format` **:** *string***) :** *string*

Parameter	Type	Description
date	*string*	a date-time representation to complete
format	*string*	the format that the `date` argument conforms to

Completes the date passed to the argument `date`, so as it conforms to the syntax of a date in CODEWORKER meaning: `"%d%b%Y %H:%M:%S.%L"`.

Starting from today date with reset time (`00:00:00.0`), it replaces date-time characteristics with those of the parameter `date` and returns the result of the substitutions.

See 4.3.84 to reading the description of a date format. A format type was added for this function: `'%|'`. Once the date has been iterated up to the end, if the format wasn't applied on it completely, an error occurs, except if `'%|'` stands at the current position in the format.

Example:

```
traceLine("Today date with reset time = '" +
completeDate(getNow(), "%d%b%Y") + "'");
local dDateAsNumber = formatDate(getNow(), "%t");
traceLine("Today date (Excel-like) = '" + dDateAsNumber + "'");
traceLine("Preceding day = '" + completeDate($dDateAsNumber -
1$, "%t") + "'");
traceLine("15th of the current month = '" + completeDate("15",
"%d") + "'");
traceLine("august of this year = '" + completeDate("08", "%m") +
"'");
traceLine("15/04 = '" + completeDate("15/04", "%d/%m") + "'");
traceLine("december 31, 2003 = '" + completeDate("december 31,
2003", "%B %d, %Y") + "'");
```

Output:

```
Today date with reset time = '28jul2008'
Today date (Excel-like) = '39657.862506'
Preceding day = '27jul2008 20:42:00.518'
15th of the current month = '15jul2008'
august of this year = '28aug2008'
15/04 = '15apr2008'
december 31, 2003 = '31dec2003'
```

See also:

`formatDate` 4.3.84, `addToDate` 4.3.3, `compareDate` 4.3.19, `getLastDelay` 4.3.95, `getNow` 4.3.96, `setNow` 4.3.185

4.3.23   completeLeftSpaces

- function **completeLeftSpaces**(`text` : *string*, `length` : *int*) : *string*

Parameter	Type	Description
text	*string*	a sequence of characters
length	*int*	the length to obtain for text

Completes the string given by argument `text` with spaces to the left, so that the resulting string takes up `length` characters long. If the argument `text` contains more than `length` characters, it returns `text`.

Example:

```
traceLine("completeLeftSpaces(1, 3) = '" + completeLeftSpaces(1,
3) + "'");
```

```
traceLine("completeLeftSpaces(123, 3) = '" +
completeLeftSpaces(123, 3) + "'");
traceLine("completeLeftSpaces(1234, 3) = '" +
completeLeftSpaces(1234, 3) + "'");
```

Output:

```
completeLeftSpaces(1, 3) = '  1'
completeLeftSpaces(123, 3) = '123'
completeLeftSpaces(1234, 3) = '1234'
```

See also:

```
countStringOccurences 4.3.34, completeRightSpaces 4.3.23, repeatString
4.3.165, replaceString 4.3.166, replaceTabulations 4.3.167, toLowerString
4.3.201, toUpperString 4.3.202, trimLeft 4.3.211, trimRight 4.3.212, trim 4.3.210,
truncateAfterString 4.3.213, truncateBeforeString 4.3.214
```

4.3.24 completeRightSpaces

- function **completeRightSpaces**(text **:** *string***,** length **:** *int***) :** *string*

Parameter	Type	Description
text	*string*	a sequence of characters
length	*int*	the length to obtain for text

Completes the string given by argument text with spaces to the right, so that the resulting string takes up length characters long. If the argument text contains more than length characters, it returns text.

Example:

```
traceLine("completeRightSpaces(1, 3) = '" + completeRightSpaces(1,
3) + "'");
traceLine("completeRightSpaces(123, 3) = '" +
completeRightSpaces(123, 3) + "'");
traceLine("completeRightSpaces(1234, 3) = '" +
completeRightSpaces(1234, 3) + "'");
```

Output:

```
completeRightSpaces(1, 3) = '1  '
completeRightSpaces(123, 3) = '123'
completeRightSpaces(1234, 3) = '1234'
```

See also:

```
countStringOccurences 4.3.34, completeLeftSpaces 4.3.22, repeatString
4.3.165, replaceString 4.3.166, replaceTabulations 4.3.167, toLowerString
4.3.201, toUpperString 4.3.202, trimLeft 4.3.211, trimRight 4.3.212, trim 4.3.210,
truncateAfterString 4.3.213, truncateBeforeString 4.3.214
```

4.3.25 composeAdaLikeString

- function **composeAdaLikeString**(text **:** *string***) :** *string*

Parameter	Type	Description
text	*string*	a sequence of character to convert to a Ada-like string

Returns the conversion of the sequence of characters given by argument text to a Ada-like string, without double quote delimiters. If text contains a double-quote, it is repeated in the sequence.

Example:

```
local sText = "double-quote \" inlayed in the sequence";
traceLine("composeAdaLikeString('" + sText + "') = '" +
composeAdaLikeString(sText) + "'");
```

Output:

```
composeAdaLikeString('double-quote " inlayed in the sequence') =
'double-quote "" inlayed in the sequence'
```

See also:

```
composeCLikeString        4.3.25,        composeHTMLLikeString        4.3.26,
composeSQLLikeString 4.3.27
```

4.3.26   composeCLikeString

- function **composeCLikeString**(text **:** *string*) **:** *string*

Parameter	Type	Description
text	*string*	a sequence of character to convert to a C-like string

Returns the conversion of the sequence of characters given by argument text to a C-like string, without double quote delimiters. It means that special characters of text are replaced by their escape sequence, the rest remaining the same.

It recognizes the following escape sequences:

- '\\' as *backslash* (\), ASCII value 92,
- '\'' as *single quotation mark* ('), ASCII value 39,
- '\"' as *double quotation mark* ("), ASCII value 34,
- '\a' as *alert* (BEL), ASCII value 7,
- '\b' as *backspace* (BS), ASCII value 8,
- '\f' as *formfeed* (FF), ASCII value 12,
- '\n' as *newline* (LF), ASCII value 10,
- '\r' as *carriage return* (CR), ASCII value 13,
- '\t' as *horizontal tab* (HT), ASCII value 9,
- '\v' as *vertical tab* (VT), ASCII value 11,

Example:

```
local sText = "\t=tabulation,\n=newline";
traceLine("composeCLikeString('" + sText + "') = '" +
composeCLikeString(sText) + "'");
```

Output:

```
composeCLikeString(' =tabulation,
=newline') = '\t=tabulation,\n=newline'
```
See also:

composeAdaLikeString 4.3.24, composeHTMLLikeString 4.3.26, composeSQLLikeString 4.3.27

4.3.27 composeHTMLLikeString

- function **composeHTMLLikeString**(text **:** *string***) :** *string*

Parameter	Type	Description
text	*string*	a sequence of character to convert to an HTML-like string

Returns the conversion of the sequence of characters given by argument text to an HTML-like string. It means that special characters of text are replaced by their HTML escape sequence (**&...;**), the rest remaining the same.

Example:
```
local sText = "< & > aren't admitted by HTML";
traceLine("composeHTMLLikeString('" + sText + "') = '" +
composeHTMLLikeString(sText) + "'");
```
Output:
```
composeHTMLLikeString('< & > aren't admitted by HTML') = '&lt;
&amp; &gt; aren&#39;t admitted by HTML'
```
See also:

composeCLikeString 4.3.25, composeAdaLikeString 4.3.24, composeSQLLikeString 4.3.27

4.3.28 composeSQLLikeString

- function **composeSQLLikeString**(text **:** *string***) :** *string*

Parameter	Type	Description
text	*string*	a sequence of character to convert to a SQL-like string

Returns the conversion of the sequence of characters given by argument text to a SQL-like string, without single quote delimiters. It means that special characters of text are replaced by their escape sequence, the rest remaining the same.

It recognizes the following escape sequences:

- '\\' as *backslash* (\), ASCII value 92,
- '\'' as *single quotation mark* ('), ASCII value 39,
- '\"' as *double quotation mark* ("), ASCII value 34,
- '\a' as *alert* (BEL), ASCII value 7,
- '\b' as *backspace* (BS), ASCII value 8,

- '**\f**' as *formfeed* (FF), ASCII value 12,
- '**\n**' as *newline* (LF), ASCII value 10,
- '**\r**' as *carriage return* (CR), ASCII value 13,
- '**\t**' as *horizontal tab* (HT), ASCII value 9,
- '**\v**' as *vertical tab* (VT), ASCII value 11,

The function translates the single quote to an escape sequence "\'", instead of repeating twice the single quote as in the SQL-standard. It presents the advantage of being more readable, but if you encounters a drawback in using this translation, apply `replaceString()` to change "\'" in "''".

Example:

```
local sText = "\t=tabulation,\n=newline,'=single quote,\"=double
quote";
traceLine("composeSQLLikeString('" + sText + "') = '" +
composeSQLLikeString(sText) + "'");
```

Output:

```
composeSQLLikeString(' =tabulation,
=newline,'=single quote,"=double quote') = '\t=tabulation,\n=newline,"=singl
quote,\"=double quote'
```

See also:

`composeCLikeString` 4.3.25, `composeAdaLikeString` 4.3.24, `composeHTMLLikeString` 4.3.26

4.3.29   computeMD5

- function **computeMD5(**`text` **:** *string***) :** *string*

Parameter	Type	Description
`text`	*string*	the string to encrypt in MD5

Computes the MD5 of a string.

This optimized MD5 implementation conforms to RFC 1321.
Source: http://www.cr0.net:8040/code/crypto/md5/
Copyright 2001-2004 Christophe Devine

Example:

```
local sSentence = "Garfield squashed 5 spiders yesterday";
local sCode = computeMD5(sSentence);
if sCode != "B2D989F0C0501E9A9D4A9F1B4D06E2C5" {
    error("bad result from 'computeMD5()'!");
}
traceLine("computeMD5('" + sSentence + "') = " + sCode);
```

Output:

```
computeMD5('Garfield squashed 5 spiders yesterday') =
B2D989F0C0501E9A9D4A9F1B4D06E2C5
```

4.3.30 copyFile

- procedure **copyFile(**sourceFileName **:** *string***,** destinationFileName **:** *string***)**

Parameter	Type	Description
sourceFileName	*string*	the name of the file to copy
destinationFileName	*string*	the name of the copy

This procedure copies a file sourceFileName to another location destinationFileName. It raises an error if something wrong has happened (either the source file doesn't exist or permissions are insufficient for copy).

Example:

```
deleteFile("Documentation/readme.txt");
copyFile("readme.txt", "Documentation/readme.txt");
traceLine("file 'readme.txt' has been copied successfully!");
```

Output:

```
file 'readme.txt' has been copied successfully!
```

See also:

appendFile 4.3.4, changeFileTime 4.3.12, chmod 4.3.16, copyGenerableFile 4.3.30, copySmartFile 4.3.32, deleteFile 4.3.45, existFile 4.3.61, fileCreation 4.3.69, fileLastAccess 4.3.70, fileLastModification 4.3.71, fileLines 4.3.72, fileMode 4.3.73, fileSize 4.3.74, loadBinaryFile 4.3.126, loadFile 4.3.127, saveBinaryToFile 4.3.171, saveToFile 4.3.174, scanFiles 4.3.176

4.3.31 copyGenerableFile

- procedure **copyGenerableFile(**sourceFileName **:** *string***,** destinationFileName **:** *string***)**

Parameter	Type	Description
sourceFileName	*string*	the name of the file to copy
destinationFileName	*string*	the name of the copy

This procedure copies a generable file sourceFileName to another location destinationFileName if the files have differences in the hand-typed text. It raises an error if something wrong has happened (either the source file doesn't exist or permissions are insufficient for copy).

A generable file is any source file containing protected areas or expandable markups.

See also:

copyFile 4.3.29, appendFile 4.3.4, changeFileTime 4.3.12, chmod 4.3.16, copySmartFile 4.3.32, deleteFile 4.3.45, existFile 4.3.61, fileCreation 4.3.69, fileLastAccess 4.3.70, fileLastModification 4.3.71, fileLines 4.3.72, fileMode 4.3.73, fileSize 4.3.74, loadBinaryFile 4.3.126, loadFile 4.3.127, saveBinaryToFile 4.3.171, saveToFile 4.3.174, scanFiles 4.3.176

4.3.32 copySmartDirectory

- procedure **copySmartDirectory**(sourceDirectory : *string*, destinationPath : *string*)

Parameter	Type	Description
sourceDirectory	*string*	the name of the directory to copy
destinationPath	*string*	the path where to copy the directory

This procedure copies a directory sourceDirectory to another location destinationPath recursively, checking for each file if it doesn't exist yet or it the content has changed. It avoids copying a file whereas it has no impact, and then modifying the last changing date of the destination file.

It raises an error if something wrong has happened (the source directory must exist and permissions must be sufficient to copy if required). Note that empty directories are ignored.

See also:

changeDirectory 4.3.11, canonizePath 4.3.9, exploreDirectory 4.3.66, getCurrentDirectory 4.3.90, relativePath 4.3.156, removeDirectory 4.3.158, resolveFilePath 4.3.168, scanDirectories 4.3.175, existDirectory 4.3.59

4.3.33 copySmartFile

- function **copySmartFile**(sourceFileName : *string*, destinationFileName : *string*) : *bool*

Parameter	Type	Description
sourceFileName	*string*	the name of the file to copy
destinationFileName	*string*	the name of the copy

This function copies a file sourceFileName to another location destinationFileName only if either file destinationFileName doesn't exist yet or the content of file destinationFileName is different of the content of file sourceFileName. It avoids copying a file when it has no impact, and then modifying the last changing date of the destination file. It raises an error if something wrong has happened (either the file doesn't exist or permissions aren't sufficient to copy when required).

If the function copies the file, and only in that case, it return true.

Example:

```
deleteFile("Documentation/readme.txt");
traceLine("First call to the 'copySmartFile()': the file is
copied");
copySmartFile("readme.txt", "Documentation/readme.txt");
traceLine("Second call to the 'copySmartFile()': nothing is
done");
copySmartFile("readme.txt", "Documentation/readme.txt");
```

Output:

```
First call to the 'copySmartFile()': the file is copied
Second call to the 'copySmartFile()': nothing is done
```

4.3.34   coreString

- function **coreString**(`text` : *string*, `pos` : *int*, `lastRemoved` : *int*) : *string*

Parameter	Type	Description
`text`	*string*	the string to work on
`pos`	*int*	the beginning position, inclusive, starting at *0*
`lastRemoved`	*int*	the number of characters to ignore at the end of `text`

Returns a substring of argument `text`. The substring begins at the position specified by argument `pos` and ignores the last characters, which number is specifier by argument `lastRemoved`. The first character starts at position *0*, the next at position *1*, and so on.

Example:

```
local sSentence = "Do you believe in human being?";
traceLine("coreString('" + sSentence + "', 18, 7) = '" +
coreString(sSentence, 18, 7) + "'");
```

Output:

```
coreString('Do you believe in human being?', 18, 7) = 'human'
```

4.3.35   countStringOccurences

- function **countStringOccurences**(`string` : *string*, `text` : *string*) : *int*

Parameter	Type	Description
`string`	*string*	sequence of characters where occurrences of substring `text` are to be counted
`text`	*string*	substring to count

Returns the number of times the substring specified by argument `text` is found into the sequence of characters held by argument `string`.

Example:

```
local sSentence = "Do you believe in human being?";
traceLine("countStringOccurences('" + sSentence + "', 'in') = "
+ countStringOccurences(sSentence, "in"));
```

Output:

```
countStringOccurences('Do you believe in human being?', 'in') =
2
```

See also:

```
completeLeftSpaces 4.3.22, completeRightSpaces 4.3.23, repeatString
4.3.165, replaceString 4.3.166, replaceTabulations 4.3.167, toLowerString
4.3.201, toUpperString 4.3.202, trimLeft 4.3.211, trimRight 4.3.212, trim 4.3.210,
truncateAfterString 4.3.213, truncateBeforeString 4.3.214
```

4.3.36   createDirectory

- function **createDirectory**(path : *string*) : *bool*

Parameter	Type	Description
path	*string*	the path of directories to create

This function creates a new directory and returns whether the operation has succeeded or not.

It fails if the complete path already exists, or if it is invalid.

4.3.37   createINETClientSocket

- function **createINETClientSocket**(remoteAddress : *string*, port : *int*) : *int*

Parameter	Type	Description
remoteAddress	*string*	a remote IP address (Internet namespace)
port	*int*	a remote port number

This function creates a client socket and connects it to the specified remote port, at the specified address IP remoteAddress, and returns a new socket descriptor. The socket is of type *stream*.

Once the creation has achieved, use directly the *send*/*receive* functions or attachInputToSocket()/attachOutputToSocket for reading/writing to the socket via a *BNF-parsing*/*template-based* script.

See also:

```
createINETServerSocket 4.3.37, acceptSocket 4.3, attachInputToSocket
4.5, detachInputFromSocket 4.5.3, attachOutputToSocket 4.6.3,
detachOutputFromSocket 4.6.7, receiveBinaryFromSocket 4.3.153,
receiveFromSocket 4.3.154, receiveTextFromSocket 4.3.155,
sendTextToSocket 4.3.180, sendBinaryToSocket 4.3.178, closeSocket 4.3.18,
flushOutputToSocket 4.6.10
```

4.3.38 createINETServerSocket

- function **createINETServerSocket(**`port` **:** *int***,** `backLog` **:** *int***) :** *int*

Parameter	Type	Description
`port`	*int*	a local port number
`backLog`	*int*	maximum queue length for incoming connection (1-5)

This function creates a server socket bound to `port` and returns a new socket descriptor. The socket is of type *stream*.

The argument `backLog` specifies the size of the queue connection. A new connection is refused when the queue is full.

Once the creation has achieved, use the function `acceptSocket()` to wait for a new client connection (blocking call).

See also:

`createINETClientSocket` 4.3.36, `acceptSocket` 4.3, `attachInputToSocket` 4.5, `detachInputFromSocket` 4.5.3, `attachOutputToSocket` 4.6.3, `detachOutputFromSocket` 4.6.7, `receiveBinaryFromSocket` 4.3.153, `receiveFromSocket` 4.3.154, `receiveTextFromSocket` 4.3.155, `sendTextToSocket` 4.3.180, `sendBinaryToSocket` 4.3.178, `closeSocket` 4.3.18, `flushOutputToSocket` 4.6.10

4.3.39 createIterator

- function **createIterator(**`i` **:** *iterator***,** `list` **:** *treeref***) :** *bool*

Parameter	Type	Description
`i`	*iterator*	iterator to initialize
`list`	*treeref*	the iterator will point to the beginning of this list

The variable `i` will become an iterator pointing to the first item of the list.

If the list is empty, there is no iterator created and the function returns `false`.

`i` must have been declared before.

Example:

```
local list = {"parsing", "tool", "and", "code", "generation"};
local it;
if !createIterator(it, list) error("shouldn't be the case!");
do {
    traceLine("\t" + it);
} while next(it);
```

Output:

```
    parsing
    tool
    and
    code
    generation
```

See also:

`createReverseIterator` 4.3.39, `duplicateIterator` 4.3.48, `first` 4.3.82, `index` 4.3.109, `last` 4.3.122, `key` 4.3.121, `next` 4.3.137, `prec` 4.3.148

4.3.40   createReverseIterator

- function **createReverseIterator**(`i` **:** *iterator***,** `list` **:** *treeref*) **:** *bool*

Parameter	Type	Description
`i`	*iterator*	iterator to initialize
`list`	*treeref*	the iterator will point to the end of this list

The variable `i` will become an iterator pointing to the last item of the list and will iterate in the reverse order.

If the list is empty, there is no iterator created and the function returns `false`.

`i` must have been declared before.

Example:

```
local list = {"parsing", "tool", "and", "code", "generation"};
local it;
if !createReverseIterator(it, list) error("shouldn't be the
case!");
do {
    traceLine("\t" + it);
} while next(it);
```

Output:

```
generation
code
and
tool
parsing
```

See also:

`createIterator` 4.3.38, `duplicateIterator` 4.3.48, `first` 4.3.82, `index` 4.3.109, `last` 4.3.122, `key` 4.3.121, `next` 4.3.137, `prec` 4.3.148

4.3.41   createVirtualFile

- function **createVirtualFile**(`handle` **:** *string***,** `content` **:** *string*) **:** *bool*

Parameter	Type	Description
`handle`	*string*	the name of the virtual file to create
`content`	*string*	the content to put into the virtual file

This function allows creating a *virtual* file. The `handle` parameter corresponds to the name given to the *virtual* file, which may be any sequence of characters. The virtual file is populated with the text assigned to the `content` argument.

The function always returns `true`, but it may be changed in the future if some naming rules will be imposed to the handles for instance. Calling the function to an existing *virtual* file causes the content to be updated with the new one.

CODEWORKER manipulates the concept of file that isn't persistent on a physical disk, but remains stored in memory. These *virtual* files may be used everywhere a file is required so as to replace it: copy, parsing or text generation. When CODEWORKER tries to open a file for reading or writing, it starts looking for a *virtual* file that has the same name.

Once a *virtual* file doesn't serve anymore, don't forget to free the useless memory it takes up by calling the `deleteVirtualFile()` function (see 4.3.46).

Example:

```
createVirtualFile("littleScript.cws",
            "a protected area:" + endl() +
            "@setProtectedArea(\"umbrella\");@finished!");
createVirtualFile("littleText.txt", "");
generate("littleScript.cws", project, "littleText.txt");
traceLine("generated (virtual) file:");
traceLine(loadVirtualFile("littleText.txt"));
deleteVirtualFile("littleText.txt");
deleteVirtualFile("littleScript.cws");
```

Output:

```
generated (virtual) file:
a protected area:
//##protect##"umbrella"
//##protect##"umbrella"
finished!
```

See also:

`createVirtualTemporaryFile` 4.3.41, `deleteVirtualFile` 4.3.46, `existVirtualFile` 4.3.63, `loadVirtualFile` 4.3.129

4.3.42   createVirtualTemporaryFile

- function **createVirtualTemporaryFile**(`content` : *string*) : *string*

Parameter	Type	Description
`content`	*string*	the content to put into the virtual file

This function allows creating a *virtual* file, for which the name of the virtual file must be chosen by the routine. The virtual file is populated with the text assigned to the `content` argument.

The function returns the name that the routine has composed for this virtual file. The only difference with the function `createVirtualFile()` (see 4.3.40) lies in the way to choose of the virtual file name. After creating the file, it behaves as any other virtual file.

Example:

```
local sScriptFile = createVirtualTemporaryFile(
            "a protected area:" + endl() +
            "@setProtectedArea(\"umbrella\");@finished!");
traceLine("Name of the (virtual) script file = '" + sScriptFile
+ "':");
```

```
local sGeneratedFile = createVirtualTemporaryFile("");
generate(sScriptFile, project, sGeneratedFile);
traceLine("generated (virtual) file '" + sGeneratedFile + "':");
traceLine(loadVirtualFile(sGeneratedFile));
deleteVirtualFile(sGeneratedFile);
deleteVirtualFile(sScriptFile);
```

Output:

```
Name of the (virtual) script file = '.˜#0':
generated (virtual) file '.˜#1':
a protected area:
//##protect##"umbrella"
//##protect##"umbrella"
finished!
```

See also:

`createVirtualFile` 4.3.40, `deleteVirtualFile` 4.3.46, `existVirtualFile` 4.3.63, `loadVirtualFile` 4.3.129

4.3.43   cutString

- procedure **cutString**(`text` : *string*, `separator` : *string*, `list` : *stringlist*)

Parameter	Type	Description
`text`	*string*	the sequence of characters to split
`separator`	*string*	the substring that separates slices
`list`	*stringlist*	the list that will contain slices

This procedure looks for slices into the argument `text`, which are separated by the substring put into argument `separator`. These slices are pushed into an array node as items called `list`.

If the argument `text` doesn't contain any occurrence of the argument `separator`, the argument `list` will contain only one item that is `text`.

Example:

```
local sText = "a yellow submarine";
local listOfItems;
traceLine("cutString('" + sText + "', ' ', listOfItems):");
cutString(sText, " ", listOfItems);
traceObject(listOfItems);
```

Output:

```
cutString('a yellow submarine', ' ', listOfItems):
Tracing variable 'listOfItems':
    ["0" -> "a", "1" -> "yellow", "2" -> "submarine"]
End of variable's trace 'listOfItems'.
```

See also:

`charAt` 4.3.13, `coreString` 4.3.33, `joinStrings` 4.3.120, `leftString` 4.3.123, `lengthString` 4.3.124, `midString` 4.3.132, `rightString` 4.3.169, `rsubString` 4.3.170, `subString` 4.3.198

4.3.44   decodeURL

- function **decodeURL**(URL **:** *string*) **:** *string*

Parameter	Type	Description
URL	*string*	readable URL to encode

Decode an URL from an HTTP request, meaning that the **'+'** character changes in a space and all hexadecimal descriptions of bytes (2 digits preceded by **'%'**) are converted to characters.

Note that conversions are transparent while doing HTTP requests.

Example:

```
local sURL = "Roger+Rabbit%26%25%24%3D%21%3F";
traceLine("URL before HTTP decoding = '" + sURL + "'");
traceLine("URL after HTTP decoding = '" + decodeURL(sURL) +
"'");
```

Output:

```
URL before HTTP decoding = 'Roger+Rabbit%26%25%24%3D%21%3F'
URL after HTTP decoding = 'Roger Rabbit&%$=!?'
```

See also:

encodeURL 4.3.49

4.3.45   decrement

- function **decrement**(number **:** *doubleref*) **:** *double*

Parameter	Type	Description
number	*doubleref*	variable to decrement

The result of decrement operation is the value of argument number *minus* one. While the result is obtained, the variable number is decremented.

Example:

```
local iNumber = 32;
traceLine("iNumber = " + iNumber);
traceLine("decrement(iNumber) = " + decrement(iNumber));
// the variable 'number' has been decremented:
traceLine("iNumber = " + iNumber);
```

Output:

```
iNumber = 32
decrement(iNumber) = 31
iNumber = 31
```

See also:

increment 4.3.107, floor 4.3.83, ceil 4.3.10

4.3.46 deleteFile

- function **deleteFile(**`filename` **:** *string***) :** *bool*

Parameter	Type	Description
`filename`	*string*	name of the file to delete

Deletes the file whose name is given by parameter `filename`. If the file cannot be found or if it cannot be deleted, the function returns `false`.

Note that if the file name is a relative path, it is understood as being relative to the current directory where the interpreter has been launched. So, the interpreter doesn't search into include directories passed to the command line (option `-I`), to offer a more secure use.

Example:

```
copyFile("readme.txt", "Documentation/readme.txt");
traceLine("Result of deleting file 'Documentation/readme.txt' =
'" + deleteFile("Documentation/readme.txt") + "'");
```

Output:

```
Result of deleting file 'Documentation/readme.txt' = 'true'
```

See also:

`copyFile` 4.3.29, `appendFile` 4.3.4, `changeFileTime` 4.3.12, `chmod` 4.3.16, `copyGenerableFile` 4.3.30, `copySmartFile` 4.3.32, `existFile` 4.3.61, `fileCreation` 4.3.69, `fileLastAccess` 4.3.70, `fileLastModification` 4.3.71, `fileLines` 4.3.72, `fileMode` 4.3.73, `fileSize` 4.3.74, `loadBinaryFile` 4.3.126, `loadFile` 4.3.127, `saveBinaryToFile` 4.3.171, `saveToFile` 4.3.174, `scanFiles` 4.3.176

4.3.47 deleteVirtualFile

- function **deleteVirtualFile(**`handle` **:** *string***) :** *bool*

Parameter	Type	Description
`handle`	*string*	the name of the virtual file to delete

This function removes from memory the *virtual* file whose name is given by the `handle` parameter.

It returns `true` if the virtual file was created before and has been removed successfully.

See also:

`createVirtualFile` 4.3.40, `createVirtualTemporaryFile` 4.3.41, `existVirtualFile` 4.3.63, `loadVirtualFile` 4.3.129

4.3.48 div

- function **div(**`dividend` **:** *double***,** `divisor` **:** *double***) :** *double*

Chapter 4. The scripting language

Parameter	Type	Description
`dividend`	*double*	the dividend
`divisor`	*double*	the divisor

Returns the result of arithmetic division `dividend` **/** `divisor`. Members are converted from strings to numbers, supposed being worth `0` if a parsing error occurs; then the division is processed, and the result is converted to a string, skipping fractional part if all digits after the dot are *0*.

Remember that the symbol **'/'** doesn't mean anything in the standard syntax of the language, so there is no way to confuse for expressing a division. However, it exists an escape mode that allows writing arithmetic expressions between **'$'** symbols, as formula under *LaTeX*. So, `$dividend / divisor$` is equivalent to `div(dividend, divisor)`.

Example:

```
local a = 3.2;
traceLine(a + " / 2 = " + div(a, "2"));
traceLine(a + " / 0.2 = " + div(a, 0.2) + " <- integer value");
```

Output:

```
3.2 / 2 = 1.6
3.2 / 0.2 = 16 <- integer value
```

See also:

`add` 4.3.1, `sub` 4.3.197, `mult` 4.3.134, `exp` 4.3.64, `log` 4.3.130, `mod` 4.3.133, `pow` 4.3.147

4.3.49   duplicateIterator

- function **duplicateIterator**(`oldIt` : *iterator*, `newIt` : *treeref*) : *bool*

Parameter	Type	Description
`oldIt`	*iterator*	the original iterator to duplicate
`newIt`	*treeref*	copy of the original iterator

Duplicates an iterator or returns `false` if *oldIt* isn't an iterator.

Example:

```
local list = {"parsing", "tool", "and", "code", "generation"};
foreach i in list {
    local it;
    if !duplicateIterator(i, it) error("shouldn't be the case!");
    traceText("\t'" + it + "' - ");
    if prec(it) traceLine("precedent value = '" + it + "'");
    else traceLine("no precedent value!");
}
```

Output:

```
    'parsing' - no precedent value!
    'tool' - precedent value = 'parsing'
    'and' - precedent value = 'tool'
    'code' - precedent value = 'and'
    'generation' - precedent value = 'code'
```

4.3.50 encodeURL

- function **encodeURL**(URL **:** *string*) **:** *string*

Parameter	Type	Description
URL	*string*	readable URL to encode

Encode an URL for an HTTP request, meaning that the space character changes in **'+'** and all non-alphanumeric characters are encoded in hexadecimal, preceded by **'%'**.

Note that conversions are transparent while doing HTTP requests.

Example:

```
local sURL = "Roger Rabbit&%$=!?";
traceLine("URL before HTTP encoding = '" + sURL + "'");
traceLine("URL after HTTP encoding = '" + encodeURL(sURL) +
"'");
```

Output:

```
URL before HTTP encoding = 'Roger Rabbit&%$=!?'
URL after HTTP encoding = 'Roger%20Rabbit%26%25%24%3D%21%3F'
```

See also:

`decodeURL` 4.3.43

4.3.51 endl

- function **endl**() **:** *string*

Returns an end of line, value depending on the platform on which the script is executed. It returns `"\r\n"` under *Windows*, and `"\n"` on any *UNIX* platform.

Example:

```
traceLine("endl() = '" + endl() + "'");
traceLine(" length = " + lengthString(endl()));
traceLine(" first ASCII character = " + charToInt(charAt(endl(),
0)));
```

Output:

```
endl() = '
'
    length = 2
    first ASCII character = 13
```

4.3.52 endString

- function **endString**(text **:** *string*, end **:** *string*) **:** *bool*

Parameter	Type	Description
text	*string*	a sequence of characters to test
end	*string*	the postfix

"true" if the argument end is a postfix of the character sequence represented by text; "" otherwise. Note also that "true" will be returned if end is an empty string or is equal to argument text.

Example:

```
local sText = "airport";
traceLine("endString('" + sText + "', 'port') = '" +
endString(sText, "port") + "'");
```

Output:

```
endString('airport', 'port') = 'true'
```

See also:

equalsIgnoreCase 4.3.54, findFirstChar 4.3.76, findLastString 4.3.78, findNextString 4.3.79, findString 4.3.81, startString 4.3.196

4.3.53 environTable

- procedure **environTable**(table **:** *tree*)

Parameter	Type	Description
table	*tree*	will contain the list of all environment variable names

The procedure returns the array of all environment variable in the argument table. The name of the environment variable is assigned to the value of the item.

Example:

```
local theTable;
environTable(theTable);
foreach i in theTable {
    if i.startString("PROCESSOR") traceLine(i);
}
```

Output:

```
PROCESSOR_ARCHITECTURE=x86
PROCESSOR_IDENTIFIER=x86 Family 15 Model 2 Stepping 7,
GenuineIntel
PROCESSOR_LEVEL=15
PROCESSOR_REVISION=0207
```

See also:

getEnv 4.3.91, existEnv 4.3.60, putEnv 4.3.150, system 4.3.200

4.3.54 equal

- function **equal**(left **:** *double***,** right **:** *double***) :** *bool*

Parameter	Type	Description
left	*double*	first number to compare
right	*double*	second number to compare

Compares two numbers and returns `true` if they are identical.

Sometimes, the operator **'=='** is suitable to compare numbers, in the case where their decimal representation are strictly the same. But be careful that expression *"7.0" == "7"* is `false`!

However, it exists an escape mode that allows writing arithmetic comparisons between **'$'** symbols, as formula under *LaTeX*. So, `$left == right$` is equivalent to `equal(left, right)`.

Example:

```
traceLine("equal(7, '7.0') = '" + equal(7, "7.0") + "'");
traceLine("7 == '7.0' = '" + (7 == "7.0") + "'");
```

Output:

```
equal(7, '7.0') = 'true'
7 == '7.0' = ''
```

See also:

`inf` 4.3.110, `sup` 4.3.199

4.3.55 equalsIgnoreCase

- function **equalsIgnoreCase**(left **:** *string***,** right **:** *string***) :** *bool*

Parameter	Type	Description
left	*string*	first string to compare
right	*string*	second string to compare

Compares two strings, ignoring the case. It returns `true` when the comparison succeeds.

Example:

```
traceLine("equalsIgnoreCase('BANANA', 'Banana') = '" +
equalsIgnoreCase("BANANA", "Banana") + "'");
traceLine("equalsIgnoreCase('BANANA', 'APPLE') = '" +
equalsIgnoreCase("BANANA", "APPLE") + "'");
```

Output:

```
equalsIgnoreCase('BANANA', 'Banana') = 'true'
equalsIgnoreCase('BANANA', 'APPLE') = ''
```

See also:

`endString` 4.3.51

4.3.56 equalTrees

- function **equalTrees(** firstTree **:** *treeref***,** secondTree **:** *treeref* **) :** *bool*

Parameter	Type	Description
firstTree	*treeref*	a parse tree
secondTree	*treeref*	another parse tree to compare with the first one

Compares two parse trees and returns `true` if they are identical (same sub-nodes, same values, same entry keys on arrays of node, repeated recursively).

Example:

```
local myTree1 = "monkey";
insert myTree1.hobbies = "to eat bretzel";
insert myTree1["Everest"] = "mountain";
insert myTree1["Tea spoon"] = "silverware";
local myTree2 = "monkey";
insert myTree2.hobbies = "to eat bretzel";
insert myTree2["Everest"] = "mountain";
insert myTree2["Tea spoon"] = "silverware";
traceLine("equalTrees(myTree1, myTree2) = '" +
equalTrees(myTree1, myTree2) + "'");
```

Output:

```
equalTrees(myTree1, myTree2) = 'true'
```

See also:

`slideNodeContent` 4.3.193

4.3.57 error

- procedure **error(** errorMessage **:** *string***)**

Parameter	Type	Description
errorMessage	*string*	an error message to throw

It raises the argument `errorMessage` as an error that may be caught into a `try/catch` statement 4.2.5.

Example:

```
try {
    error("I have forced an error!");
    traceLine("I shouldn't write this message...");
} catch(sError) {
    traceLine("I caught the error:  '" + sError + "'");
}
```

Output:

```
I caught the error:  'I have forced an error!
TEX-manual.cwt(508):  main scope
writeFunctionDescription
'
```

4.3.58   executeString

- procedure **executeString**(this **:** *tree***,** command **:** *string***)**

Parameter	Type	Description
this	*tree*	the current node that will be accessed via *this* variable
command	*string*	some instructions of the scripting language to execute

This procedure interprets the argument command as instructions to execute, written in the scripting language.

Example:

```
local myContext;
executeString(myContext,
            "traceLine(\"Beginning of string execution:\");"
            "insert this.name = \"execution\";"
            "traceLine(\"End of string execution.\");");
traceLine("What we did during the string execution:");
traceObject(myContext);
```

Output:

```
Beginning of string execution:
End of string execution.
What we did during the string execution:
Tracing variable 'myContext':
    name = "execution"
End of variable's trace 'myContext'.
```

See also:

executeStringQuiet 4.3.58

4.3.59   executeStringQuiet

- function **executeStringQuiet**(this **:** *tree***,** command **:** *string***) :** *string*

Parameter	Type	Description
this	*tree*	the current node that will be accessed with *this* variable
command	*string*	some instructions of the scripting language to execute

This function interprets the argument command as instructions to execute, written in the scripting language, but doesn't display messages to the standard output stream. Messages are put into a string that is returned by the function.

Example:

```
local myContext;
local sMessages = executeStringQuiet(myContext,
        "traceLine(\"Beginning of string execution:\");"
        "insert this.name = \"execution\";"
        "traceLine(\"End of string execution.\");");
traceLine("What we did during the execution:");
traceObject(myContext);
```

```
traceLine("What was intended to the console during the
execution:");
traceLine(sMessages);
```

Output:

```
What we did during the execution:
Tracing variable 'myContext':
    name = "execution"
End of variable's trace 'myContext'.
What was intended to the console during the execution:
Beginning of string execution:
End of string execution.
```

See also:

`executeString` 4.3.57

4.3.60   existDirectory

- function **existDirectory(**`path` **:** *string***) :** *bool*

Parameter	Type	Description
`path`	*string*	path name of the directory

The function checks the existence of a directory specified by the `path` argument.

See also:

`changeDirectory` 4.3.11, `canonizePath` 4.3.9, `copySmartDirectory` 4.3.31, `exploreDirectory` 4.3.66, `getCurrentDirectory` 4.3.90, `relativePath` 4.3.156, `removeDirectory` 4.3.158, `resolveFilePath` 4.3.168, `scanDirectories` 4.3.175

4.3.61   existEnv

- function **existEnv(**`variable` **:** *string***) :** *bool*

Parameter	Type	Description
`variable`	*string*	the environment variable name

The function returns `true` if the environment table entry contains the `variable`.

Example:

```
traceLine("PATH='" + existEnv("PATH")+ "'");
```

Output:

```
PATH='true'
```

See also:

`getEnv` 4.3.91, `environTable` 4.3.52, `putEnv` 4.3.150, `system` 4.3.200

4.3.62 existFile

- function **existFile**(fileName **:** *string*) **:** *bool*

Parameter	Type	Description
fileName	*string*	the name of a file to check for existence

Checks whether a file exists or not, looking for include directories passed on the command line.

This function doesn't work to check the existence of a directory; use `exploreDirectory()` instead.

Example:

```
local sFilename = "Documentation/CodeWorker.pdf";
traceLine("Checks existence of file '" + sFilename + "' = '" +
existFile(sFilename) + "'");
```

Output:

```
Checks existence of file 'Documentation/CodeWorker.pdf' = 'true'
```

See also:

`copyFile` 4.3.29, `appendFile` 4.3.4, `changeFileTime` 4.3.12, `chmod` 4.3.16, `copyGenerableFile` 4.3.30, `copySmartFile` 4.3.32, `deleteFile` 4.3.45, `fileCreation` 4.3.69, `fileLastAccess` 4.3.70, `fileLastModification` 4.3.71, `fileLines` 4.3.72, `fileMode` 4.3.73, `fileSize` 4.3.74, `loadBinaryFile` 4.3.126, `loadFile` 4.3.127, `saveBinaryToFile` 4.3.171, `saveToFile` 4.3.174, `scanFiles` 4.3.176

4.3.63 existVariable

- function **existVariable**(variable **:** *treeref*) **:** *bool*

Parameter	Type	Description
variable	*treeref*	the name of a variable

Checks whether a variable exists or not.

Example:

```
local alice;
traceLine("The variable 'alice' exists:  '" +
existVariable(alice) + "'");
traceLine("The variable 'wonderful' doesn't exist:  '" +
existVariable(wonderful) + "'");
```

Output:

```
(3,80):  warning!  you haven't declared the variable 'wonderful'
before ; interpreted as 'this.wonderful', but obsolete soon!
The variable 'alice' exists:  'true'
The variable 'wonderful' doesn't exist:  "
```

See also:

`clearVariable` 4.3.17, `findFirstSubstringIntoKeys` 4.3.77, `findElement` 4.3.75, `findNextSubstringIntoKeys` 4.3.80, `getArraySize` 4.3.87, `getVariableAttributes` 4.3.100, `invertArray` 4.3.114, `isEmpty` 4.3.115, `removeVariable` 4.3.164

4.3.64   existVirtualFile

- function **existVirtualFile(**`handle` **:** *string***) :** *bool*

Parameter	Type	Description
`handle`	*string*	the name of the virtual file to check

  Checks whether a *virtual* file exists or not, meaning that it has been created via the function `createVirtualFile()`.

  See also:

  `createVirtualFile` 4.3.40, `createVirtualTemporaryFile` 4.3.41, `deleteVirtualFile` 4.3.46, `loadVirtualFile` 4.3.129

4.3.65   exp

- function **exp(**`x` **:** *double***) :** *double*

Parameter	Type	Description
`x`	*double*	the floating-point whose exponential is to compute

  Returns the exponential of `x`.

  On underflow, it returns *0*.
  On overflow, it returns *infinite*.

  Example:

  `traceLine("exp(0.693147) = " + $exp(0.693147)$);`

  Output:

  `exp(0.693147) = 1.9999996388801418`

  See also:

  `add` 4.3.1, `sub` 4.3.197, `mult` 4.3.134, `div` 4.3.47, `log` 4.3.130, `mod` 4.3.133, `pow` 4.3.147

4.3.66   expand

- procedure **expand(**`patternFileName` **:** *script***,** `this` **:** *treeref***,** `outputFileName` **:** *string***)**

Parameter	Type	Description
`patternFileName`	*script<pattern>*	file name or block of instructions of the *pattern script*
`this`	*treeref*	the current node that will be accessed with *this* variable
`outputFileName`	*string*	the existing file to expand

  Expands an existing file whose name is passed to the argument `outputFileName`, by executing the *pattern script* `patternFileName` on it.

  Expanding a file consists of generating code into marked out areas only, the rest of the file remaining the same. The markup is put into a comment, knowing that the syntax of the comment must conform to the type of the expanded file `outputFileName`. So, an HTML file

expects **<!- -** and **- ->**, a JAVA file is waiting for **//** and **"\n"**, ... The markup is announced by **##markup##** followed by a string that represents the *markup key*. Don't forget to configure correctly the syntax of comment boundaries with procedures `setCommentBegin()` (see 4.3.181) and `setCommentEnd()` (see 4.3.182).

When the procedure is called, CODEWORKER jumps from a markup to another. To handle a markup, it checks whether text was already generated, put between tags **##begin##"*markup-key*"** and **##end##"*markup-key*"**, added automatically the first time an expansion is required, to demarquate the portion of code that doesn't belong to the user. Then, it extracts all protected areas, if any, and it generates code at the position of the markup, adding *begin/end* tags seen before.

If the interpreter finds the tag `##script##` just after the markup, it extracts the embedded text, considered as a script, eventually put between comments. Otherwise, the interpreter executes the *pattern script*.

Note that some data might be put between tags `##data##`, accessible in the *template-based* script via the function `getMarkupValue()` (see 4.6.14). This block of custom data comes after the `##script##` tag, if present.

The same *pattern script* is called for all markups, so, to distinguish them and not to generate always the same text, it controls the current markup key being processed via the function `getMarkupKey()` (see 4.6.13).

Be careful not to confuse this prodedure with `generate()` that doesn't care about markups and that overwrites the output file completely, except protected areas of course.

See also:

`autoexpand` 4.3.5, `generate` 4.3.85, `generateString` 4.3.86, `translate` 4.3.208, `parseAsBNF` 4.3.141, `parseFree` 4.3.142, `parseFreeQuiet` 4.3.143, `parseStringAsBNF` 4.3.144, `translateString` 4.3.209

4.3.67 exploreDirectory

- function **exploreDirectory**(`directory` : *tree*, `path` : *string*, `subfolders` : *bool*) : *bool*

Parameter	Type	Description
`directory`	*tree*	node that will contain the name of all files and folders
`path`	*string*	the directory to explore
`subfolders`	*bool*	to explore sub directories recursively

Explores the directory whose name is passed to the argument `path`. The list of files is put into the node's array `directory.`**`files`** and the list of directories are put into the node's array `directory.`**`directories`**. Exploring sub directories is required by the argument `subfolders` and each node of the node's array `directory.`**`directories`** repeats the same process recursively. The key of an array's node is the short name of the file or the directory and the value of a directory item is the relative path, whereas the value of a file item is also the short name.

If the directory cannot be found, the variable `directory` doesn't change and the function returns `false`. If the directory doesn't contain any file, the attribute `directory.`*`files`* isn't created. If the directory doesn't contain any subfolder, the attribute `directory.`*`directories`* isn't created.

Example:

```
local theDirectory;
local sPathToExplore = project.winBinaries; // Windows package
of CodeWorker
if !exploreDirectory(theDirectory, sPathToExplore, true)
error("unable to find the directory");
// the complete path is too long:  shorten it
traceLine("starting directory = '" + theDirectory.subString(sPathToExplore.l
+ "':");
foreach j in theDirectory.files {
    traceLine(" '" + j + "'");
}
foreach i in cascading theDirectory.directories {
    // the complete path is too long:  shorten it
    traceLine("- directory '" + i.subString(sPathToExplore.length())
+ "':");
    foreach j in i.directories {
        traceLine(" subfolder '" + key(j) + "'");
    // the complete path is too long:  shorten it
        traceLine(" path '" + j.subString(sPathToExplore.length())
+ "'");
    }
    if key(i) == "GettingStarted" {
        traceLine(" ...  a lot of files!");
    } else {
        foreach j in i.files {
            traceLine(" '" + j + "'");
        }
    }
}
```

Output:

```
starting directory = '/':
   'GettingStarted.bat'
   'readme.txt'
- directory '/bin/':
   'CodeWorker.exe'
   'libcurl.dll'
- directory '/include/':
   'CGCompiler.h'
   'CGExternalHandling.h'
   'CGRuntime.h'
   'CppParsingTree.h'
   'CW4dl.h'
   'DynPackage.h'
   'ExternalValueNode.h'
   'UtlException.h'
- directory '/Scripts/':
   subfolder 'Tutorial'
       path '/Scripts/Tutorial/'
- directory '/Scripts/Tutorial/':
   subfolder 'GettingStarted'
       path '/Scripts/Tutorial/GettingStarted/'
```

```
- directory '/Scripts/Tutorial/GettingStarted/':
    ... a lot of files!
```

See also:

`changeDirectory` 4.3.11, `canonizePath` 4.3.9, `copySmartDirectory` 4.3.31,
`getCurrentDirectory` 4.3.90, `relativePath` 4.3.156, `removeDirectory` 4.3.158,
`resolveFilePath` 4.3.168, `scanDirectories` 4.3.175, `existDirectory` 4.3.59

4.3.68   extendExecutedScript

- procedure **extendExecutedScript(**`scriptContent` : *string***)**

Parameter	Type	Description
`scriptContent`	*string*	A piece of CodeWorker's script to compile at the end of the currently executed script

Extend the currently executed CodeWorker's script with new instructions.   The argument
`scriptContent` is a piece of script to compile at the end of the executed script.

4.3.69   extractGenerationHeader

- function **extractGenerationHeader(**`filename` : *string***,** `generator` : *stringref***,** `version` :
*stringref***,** `date` : *stringref***)** : *string*

Parameter	Type	Description
`filename`	*string*	generated file to check
`generator`	*stringref*	name of the application that has generated the file
`version`	*stringref*	version of the generator
`date`	*stringref*	date/time of the generation

Looks for a generation header into the file passed to the argument `filename`. It returns the
comment that was put into the header (see procedure `setGenerationHeader()` 4.3.183)
during the generation, after having assigned the output parameters:

- `generator` with the name of the application that have generated the file, *"CodeWorker"*
  normally,

- `version` with the version of the generator, the version of CODEWORKER normally,

- `date` with the date and time of the generation, conforming to `"%d%b%Y %H:%M:%S"`;
  12dec2002 10:00:23 for example,

The generation header is inlayed in the comment delimiters and conforms to the format:

- if the comment holds on a single line:
  begin-comment **"##generation header##CodeWorker##"**
      version-number **"##"** *generation-date* **"##"**
      '"' comment '"' end-comment

– if the comment holds on more than one line:

```
begin-comment "##generation header##CodeWorker##"
     version-number "##" generation-date "##" end-comment
begin-comment "##header start##" end-comment
begin-comment line₁ end-comment
...
begin-comment lineₙ end-comment
begin-comment "##header end##" end-comment
```

Example:

```
setGenerationHeader("Popeye's Village\nGozo and Comino");
local sScriptFile = "GettingStarted/Tiny-JAVA.cwt";
local sFileName = "Documentation/" + project.listOfClasses#back.name
+ ".java";
generate(sScriptFile, project.listOfClasses#back, sFileName);
local sGenerator;
local sVersion;
local sDateTime;
traceLine("comment of the generation header = '" +
extractGenerationHeader(sFileName, sGenerator, sVersion,
sDateTime) + "'");
traceLine("generator = '" + sGenerator + "'");
traceLine("version = '" + sVersion + "'");
traceLine("date = '" + sDateTime + "'");
setGenerationHeader("");
```

Output:

```
comment of the generation header = 'Popeye's Village
Gozo and Comino'
generator = 'CodeWorker'
version = '3.10.4'
date = '30may2005 19:16:43'
```

See also:

`setGenerationHeader` 4.3.183, `getGenerationHeader` 4.3.92

4.3.70 fileCreation

- function **fileCreation**(`filename` **:** *string*) **:** *string*

Parameter	Type	Description
filename	*string*	name of the file to ask for its creation time

Returns the date-time of creation of file whose name is passed to the argument `filename`.

If an error occurs, it returns one code among the following:

– **"-1"**: unknown error that shouldn't appear,

– **"-2"**: permission denied,

– **"-3"**: too many files have been opened,

– **"-4"**: file not found,

Example:

```
local sFileName = "Documentation/CodeWorker.tex";
local sCreationTime = fileCreation(sFileName);
if startString(sCreationTime, "-") error("error code = " +
sCreationTime + "!");
traceLine("creation of '" + sFileName + "' = " + sCreationTime);
```

Output:

```
creation of 'Documentation/CodeWorker.tex' = 17mar2004 22:32:18
```

See also:

```
copyFile 4.3.29, appendFile 4.3.4, changeFileTime 4.3.12, chmod 4.3.16,
copyGenerableFile 4.3.30, copySmartFile 4.3.32, deleteFile 4.3.45, existFile
4.3.61, fileLastAccess 4.3.70, fileLastModification 4.3.71, fileLines 4.3.72,
fileMode 4.3.73, fileSize 4.3.74, loadBinaryFile 4.3.126, loadFile 4.3.127,
saveBinaryToFile 4.3.171, saveToFile 4.3.174, scanFiles 4.3.176
```

4.3.71   fileLastAccess

- function **fileLastAccess(**`filename` **:** *string***) :** *string*

Parameter	Type	Description
`filename`	*string*	name of the file to ask for its last access time

Returns the date-time of last access to file whose name is passed to the argument `filename`.

If an error occurs, it returns one code among the following:

- **"-1"**: unknown error that shouldn't appear,
- **"-2"**: permission denied,
- **"-3"**: too many files have been opened,
- **"-4"**: file not found,

Example:

```
local sFileName = "Documentation/CodeWorker.tex";
local sLastAccessTime = fileLastAccess(sFileName);
if startString(sLastAccessTime, "-") error("error code = " +
sLastAccessTime + "!");
traceLine("last access to '" + sFileName + "' = " +
sLastAccessTime);
```

Output:

```
last access to 'Documentation/CodeWorker.tex' = 28jul2008
00:44:32
```

See also:

```
copyFile 4.3.29, appendFile 4.3.4, changeFileTime 4.3.12, chmod 4.3.16,
copyGenerableFile 4.3.30, copySmartFile 4.3.32, deleteFile 4.3.45, existFile
4.3.61, fileCreation 4.3.69, fileLastModification 4.3.71, fileLines 4.3.72,
fileMode 4.3.73, fileSize 4.3.74, loadBinaryFile 4.3.126, loadFile 4.3.127,
saveBinaryToFile 4.3.171, saveToFile 4.3.174, scanFiles 4.3.176
```

4.3.72 fileLastModification

- function **fileLastModification(**`filename` **:** *string***) :** *string*

Parameter	Type	Description
`filename`	*string*	name of the file to ask for its last modification time

Returns the date-time of last modification to file whose name is passed to the argument `filename`.

If an error occurs, it returns one code among the following:

  - **"-1"**: unknown error that shouldn't appear,
  - **"-2"**: permission denied,
  - **"-3"**: too many files have been opened,
  - **"-4"**: file not found,

Example:

```
local sFileName = "Documentation/CodeWorker.tex";
local sLastModificationTime = fileLastModification(sFileName);
if startString(sLastModificationTime, "-") error("error code = "
+ sLastModificationTime + "!");
traceLine("last modification of '" + sFileName + "' = " +
sLastModificationTime);
```

Output:

```
last modification of 'Documentation/CodeWorker.tex' = 28jul2008
00:42:07
```

See also:

```
copyFile 4.3.29, appendFile 4.3.4, changeFileTime 4.3.12, chmod 4.3.16,
copyGenerableFile   4.3.30,   copySmartFile   4.3.32,   deleteFile   4.3.45,
existFile 4.3.61, fileCreation 4.3.69, fileLastAccess 4.3.70, fileLines
4.3.72, fileMode 4.3.73, fileSize 4.3.74, loadBinaryFile 4.3.126, loadFile
4.3.127, saveBinaryToFile 4.3.171, saveToFile 4.3.174, scanFiles 4.3.176
```

4.3.73 fileLines

- function **fileLines(**`filename` **:** *string***) :** *int*

Parameter	Type	Description
`filename`	*string*	name of the file where to count lines

Returns the number of lines that the file passed to the argument `filename` contains.

If the file cannot be found nor opened, the function returns −1.

Example:

```
local theFiles;
if !scanFiles(theFiles, "Generation", "*.cw?", true)
error("impossible to scan the directory");
local iLines = 0;
```

```
foreach i in theFiles iLines = $iLines + fileLines(i)$;
traceLine("total of script lines to generate \"CodeWorker\" = " +
iLines);
```

Output:

```
total of script lines to generate "CodeWorker" = 8708
```

See also:

```
copyFile 4.3.29, appendFile 4.3.4, changeFileTime 4.3.12, chmod 4.3.16,
copyGenerableFile 4.3.30, copySmartFile 4.3.32, deleteFile 4.3.45, existFile
4.3.61, fileCreation 4.3.69, fileLastAccess 4.3.70, fileLastModification
4.3.71, fileMode 4.3.73, fileSize 4.3.74, loadBinaryFile 4.3.126, loadFile
4.3.127, saveBinaryToFile 4.3.171, saveToFile 4.3.174, scanFiles 4.3.176
```

4.3.74 fileMode

- function **fileMode(**`filename` **:** *string***) :** *string*

Parameter	Type	Description
`filename`	*string*	file to ask for its permission setting

The `chmod` function returns the permission setting of the file specified by `filename`. The
permission setting controls *read* and *write* and *execute* access to the file. The returned value holds
the permission setting of the file as a concatenation of chars amongst the following:

- **'R'** for **r**eading permitted,
- **'W'** for **w**riting permitted,
- **'X'** for e**x**ecuting permitted (ignored on *Windows* platform),

If an error occurs, the function returns one code among the following:

- **"-1"**: unknown error that shouldn't appear,
- **"-2"**: permission denied,
- **"-3"**: too many files have been opened,
- **"-4"**: file not found,

Example:

```
local sPermission = fileMode("Documentation/CodeWorker.tex");
if startString(sPermission, "-") error("error code = " +
sPermission);
traceLine("permission on file 'Documentation/CodeWorker.tex' =
'" + sPermission + "'");
```

Output:

```
permission on file 'Documentation/CodeWorker.tex' = 'RW'
```

See also:

```
copyFile 4.3.29, appendFile 4.3.4, changeFileTime 4.3.12, chmod 4.3.16,
copyGenerableFile 4.3.30, copySmartFile 4.3.32, deleteFile 4.3.45, existFile
4.3.61, fileCreation 4.3.69, fileLastAccess 4.3.70, fileLastModification
4.3.71, fileLines 4.3.72, fileSize 4.3.74, loadBinaryFile 4.3.126, loadFile
4.3.127, saveBinaryToFile 4.3.171, saveToFile 4.3.174, scanFiles 4.3.176
```

4.3.75 fileSize

- function **fileSize**(`filename` : *string*) : *int*

Parameter	Type	Description
`filename`	*string*	name of the file to ask for its size

Returns the size of the file whose name is passed to the argument `filename`.

If an error occurs, it returns one code among the following:

- **-1**: unknown error that shouldn't appear,
- **-2**: permission denied,
- **-3**: too many files have been opened,
- **-4**: file not found,

Example:

```
local sFileName = "Documentation/CodeWorker.tex";
local iSize = fileSize(sFileName);
if isNegative(iSize) error("error code = " + iSize + "!");
traceLine("size of '" + sFileName + "' = " + iSize + "
characters");
```

Output:

```
size of 'Documentation/CodeWorker.tex' = 929422 characters
```

See also:

`copyFile` 4.3.29, `appendFile` 4.3.4, `changeFileTime` 4.3.12, `chmod` 4.3.16, `copyGenerableFile` 4.3.30, `copySmartFile` 4.3.32, `deleteFile` 4.3.45, `existFile` 4.3.61, `fileCreation` 4.3.69, `fileLastAccess` 4.3.70, `fileLastModification` 4.3.71, `fileLines` 4.3.72, `fileMode` 4.3.73, `loadBinaryFile` 4.3.126, `loadFile` 4.3.127, `saveBinaryToFile` 4.3.171, `saveToFile` 4.3.174, `scanFiles` 4.3.176

4.3.76 findElement

- function **findElement**(`value` : *string*, `variable` : *treeref*) : *bool*

Parameter	Type	Description
`value`	*string*	a key as an array entry
`variable`	*treeref*	a variable that contains an array of nodes

This function looks for a key, given by argument `value`, as an entry of the nodes array passed to argument `variable`. If the key is found, the function returns `true`, and and empty string otherwise.

Example:

```
local list;
insert list["everest"] = "everest";
insert list["karakorum"] = "karakorum";
insert list["kilimanjaro"] = "kilimanjaro";
insert list["twin peaks"] = "twin peaks";
```

```
traceLine("findElement('kilimanjaro', list) = '" +
findElement("kilimanjaro", list) + "'");
```

Output:

```
findElement('kilimanjaro', list) = 'true'
```

Method: *variable*.findElement(*value*)

See also:

```
existVariable 4.3.62, clearVariable 4.3.17, findFirstSubstringIntoKeys
4.3.77,    findNextSubstringIntoKeys    4.3.80,    getArraySize    4.3.87,
getVariableAttributes 4.3.100, invertArray 4.3.114, isEmpty 4.3.115,
removeVariable 4.3.164
```

4.3.77   findFirstChar

- function **findFirstChar**(text **:** *string***,** someChars **:** *string*) **:** *int*

Parameter	Type	Description
text	*string*	the string to explore
someChars	*string*	a set of individual characters

Returns the location into text of the first character encountered that belongs to the set of characters passed to argument someChars. The position starts counting at 0. If no occurrence has been found, the negative value −1 is returned.

Example:

```
local sText = "looking for a token:  \"...\" f(a,b) {...}";
traceLine("sText = '" + composeCLikeString(sText) + "'");
traceLine("findFirstChar(sText, '\"({') = " +
findFirstChar(sText, "\"({"));
```

Output:

```
sText = 'looking for a token:  \"...\" f(a,b) {...}'
findFirstChar(sText, '"({') = 21
```

See also:

```
endString    4.3.51,    findFirstChar    4.3.76,    findLastString    4.3.78,
findNextString 4.3.79, findString 4.3.81, startString 4.3.196
```

4.3.78   findFirstSubstringIntoKeys

- function **findFirstSubstringIntoKeys**(substring **:** *string***,** array **:** *treeref*) **:** *int*

Parameter	Type	Description
substring	*string*	a sequence of characters to search into keys of a node's array
array	*treeref*	a variable that contains an array of nodes

This function returns the position of the first item, such as its corresponding entry key into the list owned by`variable` contains the substring passed to argument `substring`. The position starts counting at `0`.

If no item is found, the negative value -1 is returned.

Example:

```
local list;
insert list["everest"] = 0;
insert list["karakorum"] = 1;
insert list["kilimanjaro"] = 2;
insert list["twin peaks"] = 3;
traceLine("findFirstSubstringIntoKeys('k', list) = " +
findFirstSubstringIntoKeys("k", list));
```

Output:

```
findFirstSubstringIntoKeys('k', list) = 1
```

See also:

`existVariable` 4.3.62, `clearVariable` 4.3.17, `findElement` 4.3.75, `findNextSubstringIntoKeys` 4.3.80, `getArraySize` 4.3.87, `getVariableAttributes` 4.3.100, `invertArray` 4.3.114, `isEmpty` 4.3.115, `removeVariable` 4.3.164

4.3.79   findLastString

- function **findLastString**(`text` **:** *string***,** `find` **:** *string***) :** *int*

Parameter	Type	Description
text	*string*	a sequence of characters to explore
find	*string*	a substring to find into `text`

Returns the position of the last occurrence of the substring `find` into the sequence of characters passed to argument `text`. The position starts counting to `0`.

If the substring `find` doesn't belong to `text`, the negative value -1 is returned.

Example:

```
local sText = "the lamp of experience";
traceLine("sText = '" + sText + "'");
traceLine("findLastString(sText, 'p') = '" +
findLastString(sText, "p") + "'");
```

Output:

```
sText = 'the lamp of experience'
findLastString(sText, 'p') = '14'
```

See also:

`findFirstChar` 4.3.76, `endString` 4.3.51, `findNextString` 4.3.79, `findString` 4.3.81, `startString` 4.3.196

4.3.80 findNextString

- function **findNextString**(text : *string*, find : *string*, position : *int*) : *int*

Parameter	Type	Description
text	*string*	a sequence of characters to explore
find	*string*	a substring to find into text
position	*int*	the position in the string (starting at 0) the search must begin

Returns the lowest beginning index of the substring find that matches the sequence of characters passed to argument text, starting the search at position included. The index starts counting to 0.

If the substring find doesn't belong to text (starting at position), the negative value −1 is returned.

Example:

```
local sText = "the lamp of experience";
traceLine("sText = '" + sText + "'");
traceLine("findNextString(sText, 'p', 8) = '" +
findNextString(sText, "p", 8) + "'");
```

Output:

```
sText = 'the lamp of experience'
findNextString(sText, 'p', 8) = '14'
```

See also:

findFirstChar 4.3.76, endString 4.3.51, findLastString 4.3.78, findString 4.3.81, startString 4.3.196

4.3.81 findNextSubstringIntoKeys

- function **findNextSubstringIntoKeys**(substring : *string*, array : *treeref*, next : *int*) : *int*

Parameter	Type	Description
substring	*string*	a sequence of characters to search into keys of a node's array
array	*treeref*	a variable that contains an array of nodes
next	*int*	the position after which looking for the next item

Returns the position of the next item of list passed to argument variable, whose entry key contains the substring given by argument substring. The next item is searched after position passed to argument next. The position starts counting at 0.

If no item is found, the negative value −1 is returned.

Example:

```
local list;
insert list["everest"] = 0;
insert list["karakorum"] = 1;
insert list["kilimanjaro"] = 2;
insert list["twin peaks"] = 3;
```

```
traceLine("findNextSubstringIntoKeys('k', list, 1) = " +
findNextSubstringIntoKeys("k", list, 1));
```

Output:

```
findNextSubstringIntoKeys('k', list, 1) = 2
```

See also:

`existVariable` 4.3.62, `clearVariable` 4.3.17, `findFirstSubstringIntoKeys` 4.3.77, `findElement` 4.3.75, `getArraySize` 4.3.87, `getVariableAttributes` 4.3.100, `invertArray` 4.3.114, `isEmpty` 4.3.115, `removeVariable` 4.3.164

4.3.82 findString

- function **findString**(`text` : *string*, `find` : *string*) : *int*

Parameter	Type	Description
`text`	*string*	a sequence of characters to explore
`find`	*string*	a substring to find into `text`

Returns the position of the first occurrence of the substring `find` into the sequence of characters passed to argument `text`. The position starts counting to `0`.

If the substring `find` doesn't belong to `text`, the negative value `-1` is returned.

Example:

```
local sText = "the lamp of experience";
traceLine("sText = '" + sText + "'");
traceLine("findString(sText, 'of') = '" + findString(sText,
"of") + "'");
```

Output:

```
sText = 'the lamp of experience'
findString(sText, 'of') = '9'
```

See also:

`findFirstChar` 4.3.76, `endString` 4.3.51, `findLastString` 4.3.78, `findNextString` 4.3.79, `startString` 4.3.196

4.3.83 first

- function **first**(`i` : *iterator*) : *bool*

Parameter	Type	Description
`i`	*iterator*	iterator of a `foreach` statement or pointing to a list

Returns `true` if the iterator argument `i` points to the first item of the iterated list.

Example:

```
local myTree;
insert myTree["Everest"] = "mountain";
insert myTree["Tea spoon"] = "silverware";
```

```
foreach i in myTree {
    if first(i) traceLine("The first item key of the list is '" +
key(i) + "'");
}
```

Output:

```
The first item key of the list is 'Everest'
```

See also:

`index` 4.3.109, `last` 4.3.122, `key` 4.3.121, `next` 4.3.137, `prec` 4.3.148, `createIterator` 4.3.38, `createReverseIterator` 4.3.39, `duplicateIterator` 4.3.48

4.3.84   floor

- function **floor**(`number` **:** *double*) **:** *int*

Parameter	Type	Description
`number`	*double*	the floating-point number to floor

Returns the largest integer that is less than or equal to `number`. If `number` isn't a number, the function returns *0*.

Example:

```
traceLine("floor(5.369e+1) = " + floor(5.369e1));
```

Output:

```
floor(5.369e+1) = 53
```

See also:

`decrement` 4.3.44, `increment` 4.3.107, `ceil` 4.3.10

4.3.85   formatDate

- function **formatDate**(`date` **:** *string***,** `format` **:** *string*) **:** *string*

Parameter	Type	Description
`date`	*string*	a date-time representation to transform
`format`	*string*	the format that will be applied to the `date` argument

Converts a date to another format, or extracts just a part of the date. The date is passed to argument `date`, and the format given by `format`.

Each field of the format specification is a single character or a format type signifying a particular *format option*. A *format option* starts with a percent sign. If a percent sign is followed by a character that has no meaning as a format type, an error is raised. To print a percent-sign character, use **'%%'**.

The format type determines how the associated argument, at the current location to the date, must be interpreted:

– **'%d'** means that a 2-digits day of the month must be written,

- – '**%e**' means that a day of the month must be written, such as *1* but not *01*
- – '**%j**' means that the day of the year must be written,
- – '**%m**' means that a 2-digits month must be written,
- – '**%B**' means that the complete english name of the month must be written,
- – '**%b**' means that the truncated english name of the month must be written: only the 3 first characters,
- – '**%Y**' means that a 4-digits year must be written,
- – '**%y**' means that a 2-digits year must be written,
- – '**%t**' means that the number of days since 30dec1899 must be written (*WingZ* format),
- – '**%w**' means that the weekday must be written as an integer (`0-6`; `0` is *sunday*),
- – '**%W**' means that the weekday must be written as the complete english name,
- – '**%H**' means that a 2-digits hour must be written,
- – '**%I**' means that a 2-digits hour (12 max) must be written,
- – '**%p**' means that `"AM"` / `"PM"` must be written,
- – '**%M**' means that a 2-digits minute must be written,
- – '**%S**' means that a 2-digits second must be written,
- – '**%L**' means that a 3-digits millisecond must be written,
- – '**%D**' is equivalent to `'%m/%d/%y'`,
- – '**%r**' is equivalent to `'%I:%M:%S %p'`,
- – '**%T**' is equivalent to `'%H:%M:%S'`,

An error occurs if a temporal argument doesn't belong to those listed above, or if the date to format doesn't conform to `"%d%b%Y %H:%M:%S.%L"`.

Example:

```
traceLine("release of the documentation = '" + getNow() + "'");
traceLine("a new format = '" + formatDate(getNow(), "%B %d, %Y")
+ "'");
traceLine("the hour only = '" + formatDate(getNow(), "%H") +
"'");
```

Output:

```
release of the documentation = '28jul2008 20:42:00.500'
a new format = 'july 28, 2008'
the hour only = '20'
```

See also:

`addToDate` 4.3.3, `compareDate` 4.3.19, `completeDate` 4.3.21, `getLastDelay` 4.3.95, `getNow` 4.3.96, `setNow` 4.3.185

4.3.86 generate

- procedure **generate**(`patternFileName` : *script*, `this` : *treeref*, `outputFileName` : *string*)

Parameter	Type	Description
patternFileName	*script<pattern>*	file name of the *pattern script*
this	*treeref*	the current node that will be accessed via *this* variable
outputFileName	*string*	the output file to generate

Generates a file whose name is passed to the argument `outputFileName`, by executing the *pattern script* `patternFileName` on it.

Up to version *2.18*, the *pattern script* was necessary passed as a script file name. Since version *2.19*, the function admits to embed the script in the place of the corresponding argument `patternFileName`, inlaying the script in brackets:

```
//generation of an HTML file, which shows the title and the
content
//of some financial market news previously extracted
generate(
{<html>
    <body>
@
foreach i in this.news {
     @@composeHTMLLikeString(i.title)@<br>@endl()@@
     @<table><tr><td>@endl()@@
       @@composeHTMLLikeString(i.body) + endl()@@
     @</td></tr></table>@endl()@@
}
@    </body>
</html>
@}, project, "news.html");
```

- It avoids the writing of 2 files, as it was unavoidable before:
  ```
  generate("news2HTML.cwt", project, "news.html");
  ```

- such as `"news2HTML.cwt"`, which contains:
  ```
  <html>
      <body>
  @
  foreach i in this.news {
       @@composeHTMLLikeString(i.title)@<br>@endl()@@
       @<table><tr><td>@endl()@@
         @@composeHTMLLikeString(i.body) + endl()@@
       @</td></tr></table>@endl()@@
  }
  @    </body>
  </html>
  ```

Generating a file consists of extracting the protected areas from the output file, before overwriting it with the text generated by the *pattern script*. It is possible to put a header of generation at the beginning of the file that will specify some information such as the name of the generating tool (CODEWORKER normally) and the version of the generator and the date of generation and a custom field of data. This header of generation (see `setGenerationHeader()` 4.3.183) isn't taken into account while comparing the new generated text with the precedent version of the file on disk.

If the output file may contain some protected areas, don't forget to configure correctly the syntax of comment boundaries with procedures `setCommentBegin()` (see 4.3.181) and `setCommentEnd()` (see 4.3.182).

Be careful not to use this prodedure instead of `expand()`. Expansion saves all text, except into markups, while generation saves protected areas only and overwrites the rest!

See also:

`expand` 4.3.65, `autoexpand` 4.3.5, `generateString` 4.3.86, `translate` 4.3.208, `parseAsBNF` 4.3.141, `parseFree` 4.3.142, `parseFreeQuiet` 4.3.143, `parseStringAsBNF` 4.3.144, `translateString` 4.3.209

4.3.87 generateString

- procedure **generateString**(`patternFileName` : *script*, `this` : *treeref*, `outputString` : *stringref*)

Parameter	Type	Description
`patternFileName`	*script<pattern>*	file name of the *pattern script*
`this`	*treeref*	the current node that will be accessed via *this* variable
`outputString`	*stringref*	the output text to generate

Generates a sequence of characters, which is stored into the argument `outputString`, by executing the *pattern script* `patternFileName` on it.

Generating a sequence of characters consists of extracting the protected areas from the `outputString` string, before overwriting it with the text generated by the *pattern script*. It is possible to put a header of generation at the beginning of the file that will specify some information such as the name of the generating tool (CODEWORKER normally) and the version of the generator and the date of generation and a custom field of data. This header of generation (see `setGenerationHeader()` 4.3.183) isn't taken into account while comparing the new generated text with the precedent version of the file on disk.

If the output string may contain some protected areas, don't forget to configure correctly the syntax of comment boundaries with procedures `setCommentBegin()` (see 4.3.181) and `setCommentEnd()` (see 4.3.182).

See also:

`expand` 4.3.65, `autoexpand` 4.3.5, `generate` 4.3.85, `translate` 4.3.208, `parseAsBNF` 4.3.141, `parseFree` 4.3.142, `parseFreeQuiet` 4.3.143, `parseStringAsBNF` 4.3.144, `translateString` 4.3.209

4.3.88 getArraySize

- function **getArraySize**(`variable` : *treeref*) : *int*

Parameter	Type	Description
`variable`	*treeref*	any node of a tree

Returns the number of items the argument `variable` contains into its embedded array, or *0* if the array doesn't exist.

Example:

```
local myTree;
insert myTree["Everest"] = "mountain";
insert myTree["Tea spoon"] = "silverware";
traceLine("getArraySize(myTree) = '" + getArraySize(myTree) +
"'");
```

Output:

```
getArraySize(myTree) = '2'
```

Method: *variable*.size()

Deprecated form: `getVariableSize` has disappeared since version *1.30*

See also:

```
existVariable 4.3.62, clearVariable 4.3.17, findFirstSubstringIntoKeys
4.3.77,    findElement    4.3.75,    findNextSubstringIntoKeys    4.3.80,
getVariableAttributes   4.3.100,   invertArray   4.3.114,   isEmpty   4.3.115,
removeVariable 4.3.164
```

4.3.89 getCommentBegin

- function **getCommentBegin() :** *string*

Returns the value of a beginning of comment, which is exploited by the procedures taking in charge the source code generation, such as `expand` or `generate`. CODEWORKER must know the format of comments recognized by the output file, to be able to extract or put protected areas, or to detect *expansion markups*.

The beginning of comment assigned by default is worth **'//'**. This is the symbol of C++ and JAVA comments that are the most frequently files encountered for generation. Use the procedure `setCommentBegin` to change it.

Some languages accept more than one format of comment. It is the case of C++ or JAVA or non-standard HTML (*Microsoft* extended HTML with the non-recommended tag *'<COMMENT'* that the *W3C* hasn't admitted). CODEWORKER can't handle more than one beginning of comment format for an output file, but you'll haven't to suffer about it, because you have the control on writing the markups into the output file, and so, to conform to a unique representation of comments.

Be careful that if the beginning and the end of comments haven't been assigned correctly before generating a file, the protected areas will not be extracted, and so, lost for ever!

Example:

```
traceLine("This example is running while processing the
documentation, so we are expecting a LaTeX comment:  '" +
getCommentBegin() + "'");
```

Output:

```
This example is running while processing the documentation, so
we are expecting a LaTeX comment:  '//'
```

See also:

```
getCommentEnd 4.3.89, setCommentBegin 4.3.181, setCommentEnd 4.3.182
```

4.3.90 getCommentEnd

- function **getCommentEnd() :** *string*

Returns the value of an end of comment, which is exploited by the procedures taking in charge the source code generation, such as `expand` or `generate`. CODEWORKER must know the format of comments recognized by the output file, to be able to extract or put protected areas, or to detect *expansion markups*.

The end of comment assigned by default is worth **'\r\n'**. This is the symbol of C++ and JAVA comments that are the most frequently files encountered for generation. Use the procedure `setCommentEnd` to change it.

Some languages accept more than one format of comment. It is the case of C++ or JAVA or non-standard HTML (*Microsoft* extended HTML with the non-recommended tag *'/COMMENT>'* that the *W3C* hasn't admitted). CODEWORKER can't handle more than one end of comment format for an output file, but you'll haven't to suffer about it, because you have the control on writing the markups into the output file, and so, to conform to a unique representation of comments.

Be careful that if the beginning and the end of comments haven't been assigned correctly before generating a file, the protected areas will not be extracted, and so, lost for ever!

Example:

```
traceLine("This example is running while processing the
documentation, so we are expecting a LaTeX comment:  '" +
composeCLikeString(getCommentEnd()) + "'");
```

Output:

```
This example is running while processing the documentation, so
we are expecting a LaTeX comment:  '\n'
```

See also:

`getCommentBegin` 4.3.88, `setCommentBegin` 4.3.181, `setCommentEnd` 4.3.182

4.3.91 getCurrentDirectory

- function **getCurrentDirectory() :** *string*

Returns the current directoryŠs name as a fully qualified path, where separators are always forward slashes **/** like in UNIX. Note that the current directory is closed by a separator.

The function returns an empty string if the path is longer than 1024 characters.

Example:

```
traceLine("current directory = '" + getCurrentDirectory() +
"'");
```

Output:

```
current directory = 'E:/projects/generator/'
```

See also:

`changeDirectory` 4.3.11, `canonizePath` 4.3.9, `copySmartDirectory` 4.3.31, `exploreDirectory` 4.3.66, `relativePath` 4.3.156, `removeDirectory` 4.3.158, `resolveFilePath` 4.3.168, `scanDirectories` 4.3.175, `existDirectory` 4.3.59

4.3.92 getEnv

- function **getEnv(**variable **:** *string***) :** *string*

Parameter	Type	Description
variable	*string*	the environment variable name

The function returns the environment table entry containing the `variable`. An error message is thrown if `variable` is not found in the environment table.

See function `existEnv()` to check the existence before getting.
Use the `putenv` function to modify the value of an environment variable.

Example:

```
traceLine("PATH='" + getEnv("PATH")+ "'");
```

Output:

```
PATH='E:\Win32App\MikTeX\miktex\bin;C:\Perl\bin\;C:\WINNT\system32;C:\WINNT;C:
Files\Fichiers communs\Adaptec Shared\System;C:\PROGRA~1\ATT\Graphviz\bin;C:\
```

See also:

`environTable` 4.3.52, `existEnv` 4.3.60, `putEnv` 4.3.150, `system` 4.3.200

4.3.93 getGenerationHeader

- function **getGenerationHeader() :** *string*

Returns the comment that is added automatically to each file generated with the procedure `generate`. Defining a comment for the generation header may be required by passing the option `-genheader` on the command line or by calling the procedure `setGenerationHeader()`.

The generation header is inlayed in the comment delimiters and conforms to the format:

- if the comment holds on a single line:
  begin-comment **"##generation header##CodeWorker##"**
      version-number **"##"** *generation-date* **"##"**
      '"' *comment* *'"'* *end-comment*
- if the comment holds on more than one line:
  begin-comment **"##generation header##CodeWorker##"**
      version-number **"##"** *generation-date* **"##"** *end-comment*
  begin-comment **"##header start##"** *end-comment*
  begin-comment *line₁* *end-comment*
  ...
  begin-comment *lineₙ* *end-comment*
  begin-comment **"##header end##"** *end-comment*

Example:

```
if !getGenerationHeader() traceLine("no generation header
required for the moment");
setGenerationHeader("Popeye's Village\nKnights of Malta");
traceLine("new generation header = '" + getGenerationHeader() +
"'");
```

```
    local sFileName = "GettingStarted/Tiny-JAVA.cwt";
    traceLine("script to execute:");
    local sContent = replaceString("\r", "", loadFile(sFileName));
    local lines;
    cutString(sContent, "\n", lines);
    foreach i in lines if !startString(i, "//")
        traceLine("\t" + i);
    traceLine("class to generate = '" + project.listOfClasses#front.name
    + "'");
    local sOutputText;
    generateString(sFileName, project.listOfClasses#front,
    sOutputText);
    traceLine("generated text:");
    traceLine(sOutputText);
    setGenerationHeader("");
```

Output:

```
no generation header required for the moment
new generation header = 'Popeye's Village
Knights of Malta'
script to execute:
    package tiny;

    public class @
    this.name@ @
    if existVariable(this.parent) {
        @ extends @this.parent.name@ @
    }
    @{
        // attributes:
    @
    function getJAVAType(myAttribute :  node) {
        local sType = myAttribute.class.name;
        if myAttribute.isArray {
            set sType = "java.util.ArrayList/*<" + sType + ">*/";
        }
        return sType;
    }

    foreach i in this.listOfAttributes {
        @ private @getJAVAType(i)@ _@i.name@ = null;
    @
    }
    @
        //constructor:
        public @this.name@() {
        }

        // accessors:
    @
    foreach i in this.listOfAttributes {
        @ public @getJAVAType(i)@ get@toUpperString(i.name)@() {
```

```
    return _@i.name@; }
        public void set@toUpperString(i.name)@(@getJAVAType(i)@
@i.name@) { _@i.name@ = @i.name@; }
    @
    }
    setProtectedArea("Methods");
    @}


class to generate = 'Planet'
generated text:
//##generation header##CodeWorker##4.5.1##28jul2008 00:44:34##
//##header start##
//Popeye's Village
//Knights of Malta
//##header end##
package tiny;

public class Planet {
    // attributes:
    private _diameter = null;

    //constructor:
    public Planet() {
    }

    // accessors:
    public getDIAMETER() { return _diameter; }
    public void setDIAMETER( diameter) { _diameter = diameter; }
//##protect##"Methods"
//##protect##"Methods"
}
```

See also:

`setGenerationHeader` 4.3.183, `extractGenerationHeader` 4.3.68

4.3.94   getHTTPRequest

- function **getHTTPRequest**(`URL` : *string*, `HTTPSession` : *tree*, `arguments` : *tree*) : *string*

Parameter	Type	Description
URL	*string*	URL of the HTTP server
HTTPSession	*tree*	an object to describe the HTTP session
arguments	*tree*	list of the arguments to GET; the key contains the name of the argument and the element gives the value

This function sends an HTTP's GET request to the HTTP server pointed to by the parameter `URL` with the list of arguments put into the the parameter `arguments`.

The function returns the document read from the HTTP server.

The function `sendHTTPRequest()` (see 4.3.179) describes the structure of the HTTP session object.

See also:

`postHTTPRequest` 4.3.146, `sendHTTPRequest` 4.3.179

4.3.95 getIncludePath

- function **getIncludePath() :** *string*

  It returns the include path passed to the command line with one or more times the setting of option `-I`, or the latest include path set via the procedure `setIncludePath()`.

  The include path is a concatenation of paths separated by semi-commas (extbf';').

  Example:

  ```
  traceLine("getIncludePath():");
  local list;
  cutString(getIncludePath(), ';', list);
  foreach i in list traceLine(i);
  ```

  Output:

  ```
  getIncludePath():
  e:\Projects\generator\Generation/
  e:\Projects\Generator/
  e:\Projects\generator\Scripts\Tutorial/
  ```

  See also:

  `getProperty`   4.3.97,   `getVersion`   4.3.101,   `getWorkingPath`   4.3.102,
  `setIncludePath`   4.3.184,   `setProperty`   4.3.186,   `setVersion`   4.3.188,
  `setWorkingPath` 4.3.189

4.3.96 getLastDelay

- function **getLastDelay() :** *double*

  The function returns the last duration that was measured by a statement modifier `delay` (see 4.2.7). The duration is expressed in seconds, eventually with a floating point.

  If the function is called during the execution while measuring the time consuming (controlling sequence under a `delay` statement modifier), it returns the time elapsed since the beginning of the time-keeping.

  Example:

  ```
  local list;
  local iIndex = 4;
  delay while isPositive(decrement(iIndex)) {
      pushItem list = "element " + iIndex;
      traceLine("creating node '" + list#back + "'");
  }
  traceLine("time of execution = " + getLastDelay() + " seconds");
  ```

  Output:

```
creating node 'element 3'
creating node 'element 2'
creating node 'element 1'
time of execution = 0.000042184132340842205 seconds
```
See also:

`formatDate` 4.3.84, `addToDate` 4.3.3, `compareDate` 4.3.19, `completeDate` 4.3.21, `getNow` 4.3.96, `setNow` 4.3.185

4.3.97 getNow

- function **getNow() :** *string*

  Returns the current date-time, conforming to the format:
  `%d%b%Y %H:%M:%S.%L`

  For explanations about *format types*, see function `formatDate` at 4.3.84.

  Example:

  `traceLine("now is '" + getNow() + "'");`

  Output:

  `now is '28jul2008 20:42:00.500'`

  Deprecated form: `today` has disappeared since version *2.09*

  See also:

  `formatDate` 4.3.84, `addToDate` 4.3.3, `compareDate` 4.3.19, `completeDate` 4.3.21, `getLastDelay` 4.3.95, `setNow` 4.3.185

4.3.98 getProperty

- function **getProperty(**`define` **:** *string***) :** *string*

Parameter	Type	Description
`define`	*string*	name of a property

  Returns the value of a property that:

    – was passed to the command line via the option `'-D'` or `'-define'`,
    – was built by the procedure `setProperty()`,

  Example:

  ```
  traceLine("getProperty('documentation') = '" +
  getProperty("documentation") + "'");
  ```

  Output:

  `getProperty('documentation') = "`

  Deprecated form: `getDefineTarget` has disappeared since version *1.30*

  See also:

  `getIncludePath` 4.3.94,   `getVersion` 4.3.101,   `getWorkingPath` 4.3.102, `setIncludePath` 4.3.184,   `setProperty` 4.3.186,   `setVersion` 4.3.188, `setWorkingPath` 4.3.189

4.3.99 getShortFilename

- function **getShortFilename(**pathFilename **:** *string***) :** *string*

Parameter	Type	Description
pathFilename	*string*	a file name with its path

Returns the short name of a file, meaning without the path. It is composed of a radical + an extension.

Example:

```
traceLine("getShortFilename('src/steakhouse\\chicken.cpp') = \""
+ getShortFilename("src/steakhouse\\chicken.cpp") + "\"");
```

Output:

```
getShortFilename('src/steakhouse\chicken.cpp') = "chicken.cpp"
```

4.3.100 getTextMode

- function **getTextMode() :** *string*

Returns the mode of text that has been retained for parsing and source code generation:

- **"DOS"**: the default value if the interpreter is running under a *Windows* platform,
- **"UNIX"**: the default value if the interpreter isn't running under a *Windows* platform,
- **"BINARY"**: not exploited yet, but intended to specify later that the parsing and the source code generation are applied on binary files,

The impact of having samp"DOS" instead of any other mode is that special comments, which announce markup keys and protected areas, will finish by "\r\n" when the end of comment is a newline '\n'.

Example:

```
traceLine("This documentation is generated under '" +
getTextMode() + "' text mode");
```

Output:

```
This documentation is generated under 'DOS' text mode
```

See also:

setTextMode 4.3.187

4.3.101 getVariableAttributes

- function **getVariableAttributes(**variable **:** *treeref***,** list **:** *tree***) :** *int*

Parameter	Type	Description
variable	*treeref*	the variable to explore
list	*tree*	will contain the name and type (reference to another node or not) of each attribute

---

Populates a list with all attribute names of a tree node. The name of branches just below the node `variable` are put into `list`.

The attribute's name is a key in the list and there is no value assigned to the item, except for attributes that point to another node (a reference). In that case, the item is worth the complete name of the referenced node.

The function returns the number of attributes found, or a negative value (-1) if the tree node `variable` doesn't exist.

Note: use `#evaluateVariable()` to navigate along a tree node, where the complete name is determined at runtime.

Example:

```
local videostores;
insert videostores["Italia"].names["Video Coliseum"].town =
"Roma";
local movies;
insert movies["Lock, Stock & Two Smoking Barrels"].director =
"Guy Ritchie";
ref movies#front.shop = videostores["Italia"].names["Video
Coliseum"];
local attributeNames;
getVariableAttributes(movies#front, attributeNames);
foreach i in attributeNames {
    if i traceLine("movies#front." + key(i) + " -> " + i);
    else traceLine("movies#front." + key(i) + " = \"" +
composeCLikeString(#evaluateVariable("movies#front." + key(i)))
+ "\"");
}
```

Output:

```
movies#front.director = "Guy Ritchie"
movies#front.shop -> videostores["Italia"].names["Video
Coliseum"]
```

See also:

`existVariable` 4.3.62, `clearVariable` 4.3.17, `findFirstSubstringIntoKeys` 4.3.77, `findElement` 4.3.75, `findNextSubstringIntoKeys` 4.3.80, `getArraySize` 4.3.87, `invertArray` 4.3.114, `isEmpty` 4.3.115, `removeVariable` 4.3.164

4.3.102 getVersion

- function **getVersion() :** *string*

Returns the *version number* of the CODEWORKER interpreter or, if a *version name* has been passed to the command line via the option `-version`, the version of old scripts being executed.

Example:

```
traceLine("The version of the interpreter is '" + getVersion() +
"'");
```

Output:

```
The version of the interpreter is '4.5.1'
```

See also:

```
getProperty    4.3.97,    getIncludePath    4.3.94,    getWorkingPath    4.3.102,
setIncludePath    4.3.184,    setProperty    4.3.186,    setVersion    4.3.188,
setWorkingPath 4.3.189
```

4.3.103 getWorkingPath

- function **getWorkingPath()** : *string*

  Returns the output directory that has been assigned to the option **-path** on the command line.

  Example:

  ```
  traceLine("'-path' = '" + getWorkingPath() + "'");
  ```

  Output:

  ```
  '-path' = 'e:\Projects\generator/'
  ```

  See also:

  ```
  getProperty    4.3.97,    getIncludePath    4.3.94,    getVersion    4.3.101,
  setIncludePath    4.3.184,    setProperty    4.3.186,    setVersion    4.3.188,
  setWorkingPath 4.3.189
  ```

4.3.104 getWriteMode

- function **getWriteMode()** : *string*

  Returns how text is written during a generation or during an implicit copy while translating: `"insert"` or `"overwrite"` mode (default mode).

  See also:

  ```
  setWriteMode 4.3.190
  ```

4.3.105 hexaToDecimal

- function **hexaToDecimal**(hexaNumber **:** *string*) **:** *int*

Parameter	Type	Description
hexaNumber	*string*	an hexadecimal integer to convert to a decimal number

  Converts an hexadecimal integer, passed to the argument hexaNumber, to a signed decimal integer and returns the result. If hexaNumber doesn't conform to the syntax of an hexadecimal number (*hexaNumber* ::= #!ignore ['**0**'..'**9**' | #noCase '**A**'..'**F**']+), the function raises an error.

  Example:

  ```
  traceLine("hexaToDecimal('FE8') = " + hexaToDecimal("FE8"));
  ```

  Output:

  ```
  hexaToDecimal('FE8') = 4072
  ```

  See also:

---

byteToChar 4.3.8, bytesToLong 4.3.6, bytesToShort 4.3.7, charToByte 4.3.14, charToInt 4.3.15, longToBytes 4.3.131, octalToDecimal 4.3.139, shortToBytes 4.3.191

4.3.106 hostToNetworkLong

- function **hostToNetworkLong(**bytes : *string***)** : *string*

Parameter	Type	Description
bytes	*string*	a 4-bytes representation of a long integer sorted in the host bytes order

Converts a 4-bytes representation of a long integer to the network bytes order. CODEWORKER stores a byte as a 2-hexadecimal digits; the function raises an error if the argument bytes is malformed.

Use longToBytes() and bytesToLong() to swap between decimal and host binary representation of a long integer.

Example:

```
traceLine("hostToNetworkLong('89ABCDEF') = '" +
hostToNetworkLong("89ABCDEF") + "'");
```

Output:

```
hostToNetworkLong('89ABCDEF') = 'EFCDAB89'
```

See also:

networkLongToHost 4.3.135, hostToNetworkShort 4.3.106, networkShortToHost 4.3.136

4.3.107 hostToNetworkShort

- function **hostToNetworkShort(**bytes : *string***)** : *string*

Parameter	Type	Description
bytes	*string*	a 2-bytes representation of a short integer sorted in the host bytes order

Converts a 2-bytes representation of a short integer to the network bytes order. CODEWORKER stores a byte as a 2-hexadecimal digits; the function raises an error if the argument bytes is malformed.

Use shortToBytes() and bytesToShort() to swap between decimal and host binary representation of a short integer.

Example:

```
traceLine("hostToNetworkShort('12EF') = '" +
hostToNetworkShort("12EF") + "'");
```

Output:

```
hostToNetworkShort('12EF') = 'EF12'
```

See also:

hostToNetworkLong 4.3.105, networkLongToHost 4.3.135, networkShortToHost 4.3.136

4.3.108 increment

- function **increment**(number : *doubleref*) : *double*

Parameter	Type	Description
number	*doubleref*	variable to increment

The result of increment operation is the value of argument number *minus* one. While the result is obtained, the variable number is incremented.

Example:

```
local iNumber = 32;
traceLine("iNumber = " + iNumber);
traceLine("increment(iNumber) = " + increment(iNumber));
// the variable 'number' has been incremented:
traceLine("iNumber = " + iNumber);
```

Output:

```
iNumber = 32
increment(iNumber) = 33
iNumber = 33
```

See also:

decrement 4.3.44, floor 4.3.83, ceil 4.3.10

4.3.109 indentFile

- function **indentFile**(file : *string*, mode : *string*) : *bool*

Parameter	Type	Description
file	*string*	name of a file to indent
mode	*string*	**default value:** *""*
		type of text to indent

Indents the file passed to parameter file, forcing the indentation mode via the argument mode. If the argument is empty or omitted, the file extension drives the indentation mode:

- *cpp*, *cxx*, *h*, *hxx*: will indent as expected for a **C++** format,
- *java*: will indent as expected for a **JAVA** format,

More format will be recognized in the future.

The function returns true if the file needed to be indented, meaning that it has changed after processing the indentation.

Example:

```
traceLine("We'll indent file 'Documentation/IndentSample.cpp'
containing:");
copyFile("Documentation/IndentSample.txt", "Documentation/IndentSample.cpp")
traceLine(loadFile("Documentation/IndentSample.cpp"));
traceLine("File changed after indenting = '" +
indentFile("Documentation/IndentSample.cpp") + "'");
```

```
traceLine("File 'Documentation/IndentSample.cpp' after
indentation:");
traceLine(loadFile("Documentation/IndentSample.cpp"));
```

Output:

```
We'll indent file 'Documentation/IndentSample.cpp' containing:
int f(int i) {
switch (i) {
    case 2:
case 3:
if (i == 2) {
h();
    }
g(i - 1);
    break;
        }
}

File changed after indenting = 'true'
File 'Documentation/IndentSample.cpp' after indentation:
int f(int i) {
    switch (i) {
        case 2:
        case 3:
            if (i == 2) {
                h();
            }
            g(i - 1);
            break;
    }
}
```

See also:

`indentText` 4.6.20

4.3.110   index

- function **index**(i : *iterator*) : *int*

Parameter	Type	Description
i	*iterator*	iterator of a `foreach` statement

Returns the position of the item the iterator points to. The position in the list begins counting at 0.

Example:

```
local myTree;
insert myTree["Everest"] = "mountain";
insert myTree["Tea spoon"] = "silverware";
foreach i in myTree {
    traceLine("The item '" + key(i) + "' is at position " +
```

```
index(i) + "");
}
```

Output:

```
The item 'Everest' is at position 0
The item 'Tea spoon' is at position 1
```

See also:

`first` 4.3.82, `last` 4.3.122, `key` 4.3.121, `next` 4.3.137, `prec` 4.3.148, `createIterator` 4.3.38, `createReverseIterator` 4.3.39, `duplicateIterator` 4.3.48

4.3.111   inf

- function **inf**(`left` : *double*, `right` : *double*) : *bool*

Parameter	Type	Description
`left`	*double*	the first member
`right`	*double*	the second member

Compares two numbers and returns `true` if the first member given by argument `left` is strictly smaller than the second member passed to argument `right`.

Don't use the operator `'<'` to compare numbers in the classical syntax of the interpreter: it only checks the lexicographical order. So, *'12 < 3'* is `true`. However, it exists an escape mode that allows writing arithmetic comparisons between **'$'** symbols, as formula under *LaTeX*. So, `$left < right$` is equivalent to `inf(left, right)`.

Example:

```
traceLine("inf(3, 12) = '" + inf(3, 12) + "'");
traceLine("3 < 12 = '" + (3 < 12) + "'");
```

Output:

```
inf(3, 12) = 'true'
3 < 12 = ''
```

See also:

`equal` 4.3.53, `sup` 4.3.199

4.3.112   inputKey

- function **inputKey**(`echo` : *bool*) : *string*

Parameter	Type	Description
`echo`	*bool*	asks for echoing the standard input on the console

Returns a character extracted from the standard input, the keyboard generally. If no key was pressed, it returns an empty string.

See statement modifiers `file_as_standard_input` (4.2.7) and `string_as_standard_input` (4.2.7) to change the source of the standard input.

If the source of the standard input is the keyboard, the argument `echo` has no effects. Otherwise, the input text is displayed into the console only if `echo` is worth `true`.

4.3.113 inputLine

- function **inputLine**(echo : *bool*, prompt : *string*) : *string*

Parameter	Type	Description
echo	*bool*	asks for echoing the standard input on the console
prompt	*string*	**default value:** *""*
		text to prompt at the beginning of the line

Returns a line that was extracted from the standard input, the keyboard generally. See statement modifiers file_as_standard_input (4.2.7) and string_as_standard_input (4.2.7) to change the source of the standard input.

If the prompt argument is populated and different of an empty string, the corresponding text is displayed at the beginning of the line.

If the source of the standard input is the keyboard, the argument echo has no effects. Otherwise, the input text is displayed into the console only if echo is worth true.

Example:

```
traceText("Please enter something> ");
local sKeyboardText = inputLine(true);
traceLine("The user said:  '" + sKeyboardText + "'");
```

Output:

```
Please enter something> These characters were typed by hand on
the keyboard!
The user said:  'These characters were typed by hand on the
keyboard!'
```

4.3.114 insertElementAt

- procedure **insertElementAt**(list : *treeref*, key : *string*, position : *int*)

Parameter	Type	Description
list	*treeref*	an array of nodes
key	*string*	the entry key of the element to insert
position	*int*	where to insert the new element, starting at 0

Insert a new element to list, at a position given by the argument position. The argument key indicates the key of this element, which is built empty.

If the key is an empty string, then the key is supposed to be worth the size of the list automatically.

You can access the new element by writing either:
list#[*position*]
or
list[*key*]

Example:

```
local list;
insert list["twin peaks"] = "twin peaks";
insert list["everest"] = "everest";
traceLine("before inserting the kilimanjaro:");
```

```
foreach i in list traceLine("\t" + i);
insertElementAt(list, "kilimanjaro", 1);
list#[1] = "kilimanjaro"; // assign a value to the new element
traceLine("after inserting the kilimanjaro at the second
place:");
foreach i in list traceLine("\t" + i);
```

Output:

```
before inserting the kilimanjaro:
    twin peaks
    everest
after inserting the kilimanjaro at the second place:
    twin peaks
    kilimanjaro
    everest
```

4.3.115   invertArray

- procedure **invertArray(**array : *treeref* **)**

Parameter	Type	Description
array	*treeref*	the array to handle

Inverts the elements of the array passed to the well-named argument array, such as the first item becomes the last one, and the last item the first one.

Example:

```
local list;
insert list["twin peaks"] = "twin peaks";
insert list["karakorum"] = "karakorum";
insert list["everest"] = "everest";
insert list["kilimanjaro"] = "kilimanjaro";
traceLine("before inverting the array:");
foreach i in list traceLine("\t" + i);
invertArray(list);
traceLine("after inverting the array:");
foreach i in list traceLine("\t" + i);
```

Output:

```
before inverting the array:
    twin peaks
    karakorum
    everest
    kilimanjaro
after inverting the array:
    kilimanjaro
    everest
    karakorum
    twin peaks
```

See also:

4.3.116   isEmpty

- function **isEmpty**(array **:** *treeref*) **:** *bool*

Parameter	Type	Description
array	*treeref*	any node of a tree

Returns **true** if the argument array embeds an array of trees, and **false** otherwise.

Example:

```
local myTree;
insert myTree["Everest"] = "mountain";
insert myTree["Tea spoon"] = "silverware";
traceLine("isEmpty(myTree) = '" + isEmpty(myTree) + "'");
```

Output:

```
isEmpty(myTree) = "
```

Method: *array*.empty()

See also:

4.3.117   isIdentifier

- function **isIdentifier**(identifier **:** *string*) **:** *bool*

Parameter	Type	Description
identifier	*string*	an identifier is a string composed of letters and underscores ; digits are admitted too, except at the first place

This predicate checks whether the string passed by parameter is an identifier or not.

Example:

```
traceLine("isIdentifier('atom') = '" + isIdentifier("atom") +
"'");
traceLine("isIdentifier('$money') = '" + isIdentifier("$money")
+ "'");
```

Output:

```
isIdentifier('atom') = 'true'
isIdentifier('$money') = "
```

4.3.118 isNegative

- function **isNegative(**number **:** *double***)** **:** *bool*

Parameter	Type	Description
`number`	*double*	a number to check

This predicate checks whether the number passed by parameter is strictly negative or not.

If the argument isn't recognized as a number, the number is supposed to be worth `0`, so the function returns `false`.

Be careful if you choose the expression `'< 0'` to compare numbers in the classical syntax of the interpreter: it only checks the lexicographical order. So, *'+0.0 <= 0'* is `false`! However, it exists an escape mode that allows writing arithmetic comparisons between **'$'** symbols, as formula under *LaTeX*. So, `$number <= 0$` is equivalent to `isNegative(number)`.

Example:

```
traceLine("isNegative(0) = '" + isNegative(0) + "'");
traceLine("isNegative(-1) = '" + isNegative(-1) + "'");
```

Output:

```
isNegative(0) = "
isNegative(-1) = 'true'
```

See also:

`isPositive` 4.3.119

4.3.119 isNumeric

- function **isNumeric(**number **:** *string***)** **:** *bool*

Parameter	Type	Description
`number`	*string*	a floating-point number in text representation

This predicate checks whether the string passed by parameter is a floating-point or not.

Example:

```
traceLine("isNumeric('atom') = '" + isNumeric("atom") + "'");
traceLine("isNumeric('3.14') = '" + isNumeric("3.14") + "'");
```

Output:

```
isNumeric('atom') = "
isNumeric('3.14') = 'true'
```

4.3.120 isPositive

- function **isPositive(**number **:** *double***)** **:** *bool*

Parameter	Type	Description
`number`	*double*	a number to check

This predicate checks whether the number passed by parameter is strictly positive or not.

If the argument isn't recognized as a number, the number is supposed to be worth `0`, so the function returns `false`.

Be careful if you choose the expression `'> 0'` to compare numbers in the classical syntax of the interpreter: it only checks the lexicographical order. So, *'-0.0 >= 0'* is `false`! However, it exists an escape mode that allows writing arithmetic comparisons between **'$'** symbols, as formula under *LaTeX*. So, `$number >= 0`$ is equivalent to `isPositive(number)`.

Example:

```
traceLine("isPositive(0) = '" + isPositive(0) + "'");
traceLine("isPositive(1) = '" + isPositive(1) + "'");
```

Output:

```
isPositive(0) = "
isPositive(1) = 'true'
```

See also:

`isNegative` 4.3.117

4.3.121 joinStrings

- function **joinStrings(**`list` **:** *tree***,** `separator` **:** *string***) :** *string*

Parameter	Type	Description
`list`	*tree*	the list that contains the strings to join
`separator`	*string*	the sequence of chars that separates the strings

This function returns the concatenation of all strings put into `list`, putting a separator between each of them.

If the list is empty, the function will return an empty string.

Example:

```
local listOfItems = {"a", "yellow", "submarine"};
traceLine("joinStrings({'a', 'yellow', 'submarine'}, './.'):");
traceLine(joinStrings(listOfItems, "./."));
```

Output:

```
joinStrings({'a', 'yellow', 'submarine'}, './.'):
a./.yellow./.submarine
```

See also:

`charAt` 4.3.13, `coreString` 4.3.33, `cutString` 4.3.42, `leftString` 4.3.123, `lengthString` 4.3.124, `midString` 4.3.132, `rightString` 4.3.169, `rsubString` 4.3.170, `subString` 4.3.198

4.3.122 key

- function **key(**`i` **:** *iterator***) :** *string*

Parameter	Type	Description
i	*iterator*	iterator of a `foreach` statement or pointing to a list

Returns the key that allows accessing the current item of the iterated list.

Example:

```
local myTree;
insert myTree["Everest"] = "mountain";
insert myTree["Tea spoon"] = "silverware";
foreach i in myTree {
    traceLine("key = '" + key(i) + "' value = '" + i + "'");
}
```

Output:

```
key = 'Everest' value = 'mountain'
key = 'Tea spoon' value = 'silverware'
```

See also:

`first` 4.3.82, `index` 4.3.109, `last` 4.3.122, `next` 4.3.137, `prec` 4.3.148, `createIterator` 4.3.38, `createReverseIterator` 4.3.39, `duplicateIterator` 4.3.48

4.3.123   last

- function **last**(i **:** *iterator*) **:** *bool*

Parameter	Type	Description
i	*iterator*	iterator of a `foreach` statement or pointing to a list

Returns `true` if the iterator argument `i` points to the last item of the iterated list.

Example:

```
local myTree;
insert myTree["Everest"] = "mountain";
insert myTree["Tea spoon"] = "silverware";
foreach i in myTree {
    if last(i) traceLine("The last item key of the list is '" +
key(i) + "'");
}
```

Output:

```
The last item key of the list is 'Tea spoon'
```

See also:

`first` 4.3.82, `index` 4.3.109, `key` 4.3.121, `next` 4.3.137, `prec` 4.3.148, `createIterator` 4.3.38, `createReverseIterator` 4.3.39, `duplicateIterator` 4.3.48

4.3.124 leftString

- function **leftString**(text : *string*, length : *int*) : *string*

Parameter	Type	Description
text	*string*	a sequence of characters
length	*int*	a positive number

Returns the first characters that belong to the string passed to the argument `text`. The number of characters to take is given by argument `length`. If the string contains less than `length` characters, the function returns all of them.

Example:

```
traceLine("leftString('airport', 3) = '" + leftString("airport",
3) + "'");
traceLine("leftString('airport', 8) = '" + leftString("airport",
8) + "'");
```

Output:

```
leftString('airport', 3) = 'air'
leftString('airport', 8) = 'airport'
```

See also:

`charAt` 4.3.13, `coreString` 4.3.33, `cutString` 4.3.42, `joinStrings` 4.3.120, `lengthString` 4.3.124, `midString` 4.3.132, `rightString` 4.3.169, `rsubString` 4.3.170, `subString` 4.3.198

4.3.125 lengthString

- function **lengthString**(text : *string*) : *int*

Parameter	Type	Description
text	*string*	a sequence of characters

Returns the length of the sequence of characters represented by argument `text`.

Example:

```
local sText = "A rabbit ran in the garden"; // size of this
string is 26 characters
traceLine("lengthString(\"" + sText + "\") = " +
lengthString(sText));
```

Output:

```
lengthString("A rabbit ran in the garden") = 26
```

Method: *text*.length()

See also:

`charAt` 4.3.13, `coreString` 4.3.33, `cutString` 4.3.42, `joinStrings` 4.3.120, `leftString` 4.3.123, `midString` 4.3.132, `rightString` 4.3.169, `rsubString` 4.3.170, `subString` 4.3.198

4.3.126 listAllGeneratedFiles

- procedure **listAllGeneratedFiles(**`files` **:** *treeref***)**

Parameter	Type	Description
`files`	*treeref*	populated with the names of all files generated since the interpreter has launched

Populates the parameter `files` with the list of all output files generated since the interpreter has launched.

The array `files` indexes each node with the name of the generated output file, and each node owns a branch called `scripts`.
This branch gives the list of all template-based scripts that have contributed to the generation of the output file (often one script only, but could be more).
The key index and the value of the nodes in the array `scripts` are worth the script file names.

The procedure raises an error if the tree parameter `files` doesn't exist.

Example:

```
local allOutputFiles;
listAllGeneratedFiles(allOutputFiles);
traceLine("List of all generated files:");
foreach i in allOutputFiles {
    // A lot of output files are generated before building
    // this document, such as C++ sources of CodeWorker:
    // they are ignored
    if i.endString(".cpp") || i.endString(".h") continue;
    // Other files are displayed:
    traceLine(" * '" + i.key() + "'");
    traceText(" -> {");
    foreach j in i.scripts {
        if !j.first() traceText(", ");
        traceText('\"' + j + '\"');
    }
    traceLine('}');
}
```

Output:

```
List of all generated files:
    * '.#f2'
        -> {"e:/Projects/generator/Generation/LaTeX2HTML.cwp"}
    * 'e:/Projects/generator/Documentation/GeneratingExamples.cwt'
        -> {"e:/Projects/Generator/Documentation/GeneratingExamplesBuilder.cw
    * 'e:/Projects/generator/Documentation/ParsingExamples.cws'
        -> {"e:/Projects/Generator/Documentation/ParsingExamplesBuilder.cwt"}
    * 'e:/Projects/generator/Scripts/Tutorial/GettingStarted/JAVA/solarsystem
        -> {"e:/Projects/generator/Scripts/Tutorial/GettingStarted/JAVAObject
    * 'e:/Projects/generator/Scripts/Tutorial/GettingStarted/JAVA/solarsystem
        -> {"e:/Projects/generator/Scripts/Tutorial/GettingStarted/JAVAObject
    * 'e:/Projects/generator/Scripts/Tutorial/GettingStarted/JAVA/solarsystem
        -> {"e:/Projects/generator/Scripts/Tutorial/GettingStarted/JAVAObject
    * 'e:/Projects/generator/Scripts/Tutorial/GettingStarted/SolarSystem.text
        -> {"e:/Projects/generator/Scripts/Tutorial/GettingStarted/HTML2LaTeX
```

```
        * 'e:/Projects/generator/Scripts/Tutorial/GettingStarted/SolarSystem0.htm
            -> {"e:/Projects/generator/Scripts/Tutorial/GettingStarted/HTMLDocume
        * 'e:/Projects/generator/Scripts/Tutorial/GettingStarted/SolarSystem1.htm
            -> {"e:/Projects/generator/Scripts/Tutorial/GettingStarted/HTMLDocume
        * 'e:/Projects/generator/WebSite/ScriptsRepository.html'
            -> {"e:/Projects/generator/Generation/WebSite.cwt"}
        * 'e:/Projects/generator/WebSite/repository/CodeWorker_grammar.cwp'
            -> {"e:/Projects/generator/Generation/CWgrammar_expander.cwt"}
        * 'e:/projects/generator/Documentation/SolarSystem.java'
            -> {"e:/Projects/generator/Scripts/Tutorial/GettingStarted/Tiny-JAVA.
        * 'e:/projects/generator/Scripts/Tutorial/GettingStarted/Tiny0.html'
            -> {"e:/Projects/Generator/Scripts/Tutorial/GettingStarted/Tiny-HTML.
        * 'e:/projects/generator/Scripts/Tutorial/GettingStarted/tiny/A.java'
            -> {"e:/Projects/Generator/Scripts/Tutorial/GettingStarted/Tiny-JAVA.
        * 'e:/projects/generator/Scripts/Tutorial/GettingStarted/tiny/B.java'
            -> {"e:/Projects/Generator/Scripts/Tutorial/GettingStarted/Tiny-JAVA.
        * 'e:/projects/generator/Scripts/Tutorial/GettingStarted/tiny/C.java'
            -> {"e:/Projects/Generator/Scripts/Tutorial/GettingStarted/Tiny-JAVA.
        * 'e:/projects/generator/Scripts/Tutorial/GettingStarted/tiny/D.java'
            -> {"e:/Projects/Generator/Scripts/Tutorial/GettingStarted/Tiny-JAVA.
        * 'e:/projects/generator/WebSite/examples/cdcatalog.cwt'
            -> {"e:/projects/generator/WebSite/repository/GenBeautifier.cwp",
    "e:/projects/generator/WebSite/repository/XSLtoCodeWorker.cwt"}
        * 'e:/projects/generator/WebSite/examples/cdcatalog.html'
            -> {"e:/projects/generator/WebSite/examples/cdcatalog.cwt"}
        * 'e:/projects/generator/WebSite/examples/ejb-jar_2_0-parser.cwp'
            -> {"e:/projects/generator/WebSite/repository/DTDtoBNF.cwt"}
        * 'e:/projects/generator/WebSite/highlighting/CWML.html'
            -> {"e:/Projects/Generator/WebSite/repository/CWscript2HTML.cwp"}
        * 'e:/projects/generator/WebSite/highlighting/CWscript2HTML.html'
            -> {"e:/Projects/Generator/WebSite/repository/CWscript2HTML.cwp"}
        * 'e:/projects/generator/WebSite/highlighting/CodeWorker_grammar.html'
            -> {"e:/Projects/Generator/WebSite/repository/CWscript2HTML.cwp"}
        * 'e:/projects/generator/WebSite/highlighting/DTDparser.html'
            -> {"e:/Projects/Generator/WebSite/repository/CWscript2HTML.cwp"}
        * 'e:/projects/generator/WebSite/highlighting/DTDtoBNF-example1.html'
            -> {"e:/Projects/Generator/WebSite/repository/CWscript2HTML.cwp"}
        * 'e:/projects/generator/WebSite/highlighting/DTDtoBNF.html'
            -> {"e:/Projects/Generator/WebSite/repository/CWscript2HTML.cwp"}
        * 'e:/projects/generator/WebSite/highlighting/RawProfiling-example1.html
            -> {"e:/Projects/Generator/WebSite/repository/CWscript2HTML.cwp"}
        * 'e:/projects/generator/WebSite/highlighting/RawProfilingCpp.html'
            -> {"e:/Projects/Generator/WebSite/repository/CWscript2HTML.cwp"}
        * 'e:/projects/generator/WebSite/highlighting/RawProfilingCppTransformat
            -> {"e:/Projects/Generator/WebSite/repository/CWscript2HTML.cwp"}
        * 'e:/projects/generator/WebSite/highlighting/RawProfilingHpp.html'
            -> {"e:/Projects/Generator/WebSite/repository/CWscript2HTML.cwp"}
        * 'e:/projects/generator/WebSite/highlighting/RawProfilingLeader.html'
            -> {"e:/Projects/Generator/WebSite/repository/CWscript2HTML.cwp"}
        * 'e:/projects/generator/WebSite/highlighting/XMLparser-example1.html'
            -> {"e:/Projects/Generator/WebSite/repository/CWscript2HTML.cwp"}
```

```
* ’e:/projects/generator/WebSite/highlighting/XMLparser.html’
     -> {"e:/Projects/Generator/WebSite/repository/CWscript2HTML.cwp"}
* ’e:/projects/generator/WebSite/highlighting/XSLparser.html’
     -> {"e:/Projects/Generator/WebSite/repository/CWscript2HTML.cwp"}
* ’e:/projects/generator/WebSite/highlighting/XSLtoBNF-example1.html’
     -> {"e:/Projects/Generator/WebSite/repository/CWscript2HTML.cwp"}
* ’e:/projects/generator/WebSite/highlighting/XSLtoCodeWorker.html’
     -> {"e:/Projects/Generator/WebSite/repository/CWscript2HTML.cwp"}
* ’e:/projects/generator/WebSite/highlighting/basicInformation.html’
     -> {"e:/Projects/Generator/WebSite/repository/CWscript2HTML.cwp"}
* ’e:/projects/generator/WebSite/highlighting/classDiagram.html’
     -> {"e:/Projects/Generator/WebSite/repository/CWscript2HTML.cwp"}
* ’e:/projects/generator/WebSite/highlighting/classDiagramGraphViz.html’
     -> {"e:/Projects/Generator/WebSite/repository/CWscript2HTML.cwp"}
* ’e:/projects/generator/WebSite/highlighting/hitCounter.html’
     -> {"e:/Projects/Generator/WebSite/repository/CWscript2HTML.cwp"}
* ’e:/projects/generator/WebSite/highlighting/hitCounterParser.html’
     -> {"e:/Projects/Generator/WebSite/repository/CWscript2HTML.cwp"}
* ’e:/projects/generator/WebSite/highlighting/hitCounterUpdate.html’
     -> {"e:/Projects/Generator/WebSite/repository/CWscript2HTML.cwp"}
```

4.3.127   loadBinaryFile

- function **loadBinaryFile**(`file` : *string*, `length` : *int*) : *string*

Parameter	Type	Description
`file`	*string*	name of the binary file to load
`length`	*int*	**default value:** *-1* number of bytes to read

Returns the binary content of the file whose name is passed to argument `file`, or the *length* first bytes only if this facultative argument isn't negative. The content concatenates a sequence of hexadecimal digits, so a byte is stored in 2 characters:

binary-content ::= [*byte*]*;
byte ::= [**'0'..'9'** | **'A'..'F'** | **'a'..'f'**]2;

If the file doesn't exist or can't be read with success, an error occurs.

Example:

```
local sContent = loadBinaryFile("readme.txt");
local sFormatedContent;
local iLine = 0;
while sContent && $iLine < 10$ {
   sFormatedContent += leftString(sContent, 40) + endl();
   sContent = sContent.subString(40);
   increment(iLine);
}
traceLine("the first 200 bytes of ’readme.txt’ are:" + endl() +
sFormatedContent);
```

Output:

```
the first 200 bytes of 'readme.txt' are:
2F2F2F2F2F2F2F2F2F2F2F2F2F2F2F2F2F2F2F2F
2F2F2F2F2F2F2F2F2F2F2F2F2F2F2F2F2F2F2F2F
2F2F2F2F2F2F2F2F2F2F0D0A2F2F202020202020
202020202020202020202020436F6465576F72
6B6572202020202020202020202020202020202020
2F2F0D0A2F2F2020202020202020202020202020
202020202D2D2D2D2D2D2D2D2D2D2020202020
202020202020202020202020202F2F0D0A2F2F2F2F
2F2F2F2F2F2F2F2F2F2F2F2F2F2F2F2F2F2F2F2F
2F2F2F2F2F2F2F2F2F2F2F2F2F2F2F2F2F2F2F2F
```

See also:

`copyFile` 4.3.29, `appendFile` 4.3.4, `changeFileTime` 4.3.12, `chmod` 4.3.16, `copyGenerableFile` 4.3.30, `copySmartFile` 4.3.32, `deleteFile` 4.3.45, `existFile` 4.3.61, `fileCreation` 4.3.69, `fileLastAccess` 4.3.70, `fileLastModification` 4.3.71, `fileLines` 4.3.72, `fileMode` 4.3.73, `fileSize` 4.3.74, `loadFile` 4.3.127, `saveBinaryToFile` 4.3.171, `saveToFile` 4.3.174, `scanFiles` 4.3.176

4.3.128 loadFile

- function **loadFile**(`file` : *string*, `length` : *int*) : *string*

Parameter	Type	Description
`file`	*string*	name of the file to load
`length`	*int*	**default value:** *-1*
		number of characters to read

Returns the content of the file whose name is passed to argument `file`, or the *length* first characters only if this facultative argument isn't negative.

If the file doesn't exist or couldn't be read with success, an error occurs.

Example:

```
local sText = loadFile("readme.txt");
sText = sText.leftString(200);
traceLine("the 200 first characters of 'readme.txt' are:" +
endl() + sText);
```

Output:

```
the 200 first characters of 'readme.txt' are:
/////////////////////////////////////////
// CodeWorker //
// ------- //
/////////////////////////////////////
```

See also:

`copyFile` 4.3.29, `appendFile` 4.3.4, `changeFileTime` 4.3.12, `chmod` 4.3.16, `copyGenerableFile` 4.3.30, `copySmartFile` 4.3.32, `deleteFile` 4.3.45, `existFile` 4.3.61, `fileCreation` 4.3.69, `fileLastAccess` 4.3.70, `fileLastModification` 4.3.71, `fileLines` 4.3.72, `fileMode` 4.3.73, `fileSize` 4.3.74, `loadBinaryFile` 4.3.126, `saveBinaryToFile` 4.3.171, `saveToFile` 4.3.174, `scanFiles` 4.3.176

4.3.129   loadProject

- procedure **loadProject(**`XMLorTXTFileName` **:** *string*, `nodeToLoad` **:** *tree***)**

Parameter	Type	Description
`XMLorTXTFileName`	*string*	an input file whose content describes a node
`nodeToLoad`	*tree*	**default value:** *project*
		the node to populate from the file; if omitted, it is defaulted to the global variable `project`

Loads a parse tree previously saved thanks to **saveProject()**.

See also:

`saveProject` 4.3.172, `saveProjectTypes` 4.3.173

4.3.130   loadVirtualFile

- function **loadVirtualFile(**`handle` **:** *string***) :** *string*

Parameter	Type	Description
`handle`	*string*	the name of the virtual file to load

Returns the content of the *virtual* file whose name is passed to argument `file`.

If the *virtual* file doesn't exist or couldn't be read with success, an error occurs.

See also:

`createVirtualFile` 4.3.40, `createVirtualTemporaryFile` 4.3.41, `deleteVirtualFile` 4.3.46, `existVirtualFile` 4.3.63

4.3.131   log

- function **log(**`x` **:** *double***) :** *double*

Parameter	Type	Description
`x`	*double*	the floating-point whose logarithm is to compute

Returns the logarithm of `x`.

If `x` is negative, it throws an error.
If `x` is 0, it returns *infinite*.

Example:

```
traceLine("log(5.369e+14)/log(10) = " + $log(5.369e+14)/log(10)$);
traceLine("log(0) = " + log(0));
```

Output:

```
log(5.369e+14)/log(10) = 14.729893403963237
log(0) = -1.#INF00000000e+000
```

See also:

`add` 4.3.1, `sub` 4.3.197, `mult` 4.3.134, `div` 4.3.47, `exp` 4.3.64, `mod` 4.3.133, `pow` 4.3.147

4.3.132 longToBytes

- function **longToBytes**(`long` **:** *ulong*) **:** *string*

Parameter	Type	Description
`long`	*ulong*	an unsigned long integer using the decimal base

Converts an unsigned long integer in decimal base to its 4-bytes representation. Bytes are ordered in the host order (memory storage).

Example:

```
traceLine("longToBytes(65535) = '" + longToBytes(65535) + "'");
```

Output:

```
longToBytes(65535) = 'FFFF0000'
```

See also:

`byteToChar` 4.3.8, `bytesToLong` 4.3.6, `bytesToShort` 4.3.7, `charToByte` 4.3.14, `charToInt` 4.3.15, `hexaToDecimal` 4.3.104, `octalToDecimal` 4.3.139, `shortToBytes` 4.3.191

4.3.133 midString

- function **midString**(`text` **:** *string***,** `pos` **:** *int***,** `length` **:** *int*) **:** *string*

Parameter	Type	Description
`text`	*string*	a sequence of characters
`pos`	*int*	a position into argument `text`
`length`	*int*	the number of characters to extract

Returns a substring located into the string `text` to the position passed to the argument `pos`. The position starts counting at `0`. The substring will be extracted for a size given by parameter `length`, or less if it has reached the end of the string.

If the argument `pos` is greater than the length of `text`, the function returns an empty string.

Example:

```
local sText = "Banks offer weapons without bullets";
traceLine("midString('" + sText + "', 12, 7) = '" +
midString(sText, 12, 7) + "'");
```

Output:

```
midString('Banks offer weapons without bullets', 12, 7) =
'weapons'
```

See also:

`charAt` 4.3.13, `coreString` 4.3.33, `cutString` 4.3.42, `joinStrings` 4.3.120, `leftString` 4.3.123, `lengthString` 4.3.124, `rightString` 4.3.169, `rsubString` 4.3.170, `subString` 4.3.198

4.3.134 mod

- function **mod**(dividend **:** *int***,** divisor **:** *int*) **:** *int*

Parameter	Type	Description
dividend	*int*	the first operand
divisor	*int*	the second operand

Returns the remainder when the first operand is divided by the second. It applies the modulus operator. Members are converted from strings to integers, supposed being worth 0 if a parsing error occurs; then the modulus is processed, and the result is converted to a string.

Remember that the symbol **'%'** doesn't mean anything in the standard syntax of the language, so there is no way to confuse for expressing a modulus operator. However, it exists an escape mode that allows writing arithmetic expressions between **'$'** symbols, as formulae under *LaTeX*. So, `$dividend % divisor$` is equivalent to `mod(dividend, divisor)`.

Example:

```
traceLine("mod(5, 2) = '" + mod(5, 2) + "'");
```

Output:

```
mod(5, 2) = '1'
```

See also:

add 4.3.1, sub 4.3.197, mult 4.3.134, div 4.3.47, exp 4.3.64, log 4.3.130, pow 4.3.147

4.3.135 mult

- function **mult**(left **:** *double***,** right **:** *double*) **:** *double*

Parameter	Type	Description
left	*double*	the first operand
right	*double*	the second operand

Returns the result of arithmetic multiplication `left * right`. Members are converted from strings to numbers, supposed being worth 0 if a parsing error occurs; then the multiplication is processed, and the result is converted to a string, skipping fractional part if all digits after the dot are *0*.

Remember that the symbol **'*'** doesn't mean anything in the standard syntax of the language, so there is no way to confuse for expressing a multiplication. However, it exists an escape mode that allows writing arithmetic expressions between **'$'** symbols, as formulae under *LaTeX*. So, `$left * right$` is equivalent to `mult(left, right)`.

Example:

```
traceLine("mult(5.5, 2) = '" + mult(5.5, 2) + "'");
```

Output:

```
mult(5.5, 2) = '11'
```

See also:

add 4.3.1, sub 4.3.197, div 4.3.47, exp 4.3.64, log 4.3.130, mod 4.3.133, pow 4.3.147

4.3.136 networkLongToHost

- function **networkLongToHost**(`bytes` : *string*) : *string*

Parameter	Type	Description
`bytes`	*string*	a 4-bytes representation of a long integer sorted in the network bytes order

Converts a 4-bytes representation of a long integer to the host bytes order. CODEWORKER stores a byte as a 2-hexadecimal digits; the function raises an error if the argument `bytes` is malformed.

Use `longToBytes()` and `bytesToLong()` to swap between decimal and host binary representation of a long integer.

Example:

```
traceLine("networkLongToHost('EFCDAB89') = '" +
networkLongToHost("EFCDAB89") + "'");
```

Output:

```
networkLongToHost('EFCDAB89') = '89ABCDEF'
```

See also:

`hostToNetworkLong` 4.3.105, `hostToNetworkShort` 4.3.106, `networkShortToHost` 4.3.136

4.3.137 networkShortToHost

- function **networkShortToHost**(`bytes` : *string*) : *string*

Parameter	Type	Description
`bytes`	*string*	a 2-bytes representation of a short integer sorted in the network bytes order

Converts a 2-bytes representation of a short integer to the host bytes order. CODEWORKER stores a byte as a 2-hexadecimal digits; the function raises an error if the argument `bytes` is malformed.

Use `shortToBytes()` and `bytesToShort()` to swap between decimal and host binary representation of a short integer.

Example:

```
traceLine("networkShortToHost('EF12') = '" +
networkShortToHost("EF12") + "'");
```

Output:

```
networkShortToHost('EF12') = '12EF'
```

See also:

`hostToNetworkLong` 4.3.105, `networkLongToHost` 4.3.135, `hostToNetworkShort` 4.3.106

4.3.138 next

- function **next**(i : *iterator*) : *bool*

Parameter	Type	Description
i	*iterator*	iterator of a `foreach` statement or pointing to items of a list

The iterator will now point to the next item of the list and returns `true` if exists.

See also:

`first` 4.3.82, `index` 4.3.109, `last` 4.3.122, `key` 4.3.121, `prec` 4.3.148, `createIterator` 4.3.38, `createReverseIterator` 4.3.39, `duplicateIterator` 4.3.48

4.3.139 not

- function **not**(expression : *bool*) : *bool*

Parameter	Type	Description
expression	*bool*	any kind of expression

This function does the same work as the unary operator '**!**': it returns `true` if the evaluation of the `expression` is an empty string, and `false` otherwise.

Example:

```
local myVariable;
traceLine("not(existVariable(myVariable)) = '" +
not(existVariable(myVariable)) + "'");
```

Output:

```
not(existVariable(myVariable)) = "
```

4.3.140 octalToDecimal

- function **octalToDecimal**(octalNumber : *string*) : *int*

Parameter	Type	Description
octalNumber	*string*	an octal integer to convert to a decimal number

Converts an octal integer, passed to the argument `octalNumber`, to a signed decimal integer and returns the result. If `octalNumber` doesn't conform to the syntax of an octal number (*octalNumber* ::= #!ignore [**'0'..'8'**]+), the function raises an error.

Example:

```
traceLine("octalToDecimal('765') = " + octalToDecimal("765"));
```

Output:

```
octalToDecimal('765') = 501
```

See also:

`byteToChar` 4.3.8, `bytesToLong` 4.3.6, `bytesToShort` 4.3.7, `charToByte` 4.3.14, `charToInt` 4.3.15, `hexaToDecimal` 4.3.104, `longToBytes` 4.3.131, `shortToBytes` 4.3.191

4.3.141 openLogFile

- procedure **openLogFile**(`filename` **:** *string*)

Parameter	Type	Description
`filename`	*string*	name of the file where log information will be put

Creates (or erases if already exists) a log file, which remains valid upto the end of the execution.
Each *trace* function (`traceLine()`, `traceText()`, `traceStack()`) will write in the log file.
This function is very convenient for debugging a CGI script, where the standard output is devoted to the result page.

Note that passing an empty filename stops the log mechanism.

4.3.142 parseAsBNF

- procedure **parseAsBNF**(`BNFFileName` **:** *script*, `this` **:** *tree*, `inputFileName` **:** *string*)

Parameter	Type	Description
`BNFFileName`	*script<BNF>*	the name of the *BNF-driven* parsing script
`this`	*tree*	the current node that will be accessed with `this` variable
`inputFileName`	*string*	the file to parse

Parses an input file whose name is given by the argument `inputFileName`. It executes the `BNF-driven` script called `BNFFileName`; see section 4.3.216 for more information.

See also:

`parseFree` 4.3.142, `parseFreeQuiet` 4.3.143, `parseStringAsBNF` 4.3.144, `translate` 4.3.208, `translateString` 4.3.209, `expand` 4.3.65, `autoexpand` 4.3.5, `generate` 4.3.85, `generateString` 4.3.86

4.3.143 parseFree

- procedure **parseFree**(`designFileName` **:** *script*, `this` **:** *tree*, `inputFileName` **:** *string*)

Parameter	Type	Description
`designFileName`	*script<free>*	the name of the parsing script that reads tokens in a procedural way
`this`	*tree*	the current node that will be accessed with *this* variable
`inputFileName`	*string*	the file to parse

Parses an input file whose name is given by the argument `inputFileName`. It executes the `procedural-driven` script called `designFileName`; see section 4.4.6 for more information.

Deprecated form: `loadDesign` has disappeared since version *1.6*

See also:

`parseAsBNF` 4.3.141, `parseFreeQuiet` 4.3.143, `parseStringAsBNF` 4.3.144, `translate` 4.3.208, `translateString` 4.3.209, `expand` 4.3.65, `autoexpand` 4.3.5, `generate` 4.3.85, `generateString` 4.3.86

4.3.144 parseFreeQuiet

- function **parseFreeQuiet**(designFileName **:** *string*, this **:** *tree*, inputFileName **:** *string*) **:** *string*

Parameter	Type	Description
designFileName	*string*	the name of the parsing script that reads tokens in a procedural way
this	*tree*	the current node that will be accessed with *this* variable
inputFileName	*string*	the file to parse

This function parses the file passed to argument inputFileName, following the instructions of the procedure-driven parsing script given by parameter designFileName, but doesn't display messages to the standard output stream. Messages are put into a string that is returned by the function.

Example:

```
local sScript = "GettingStarted/SimpleML-token-reading.cws";
local sDesign = "GettingStarted/SolarSystem0.sml";
traceLine("sScript = '" + sScript + "'");
traceLine("sDesign = '" + sDesign + "'");
traceLine("messages of parseFreeQuiet(sScript, project,
sDesign):");
traceLine(parseFreeQuiet(sScript, project, sDesign));
```

Output:

```
sScript = 'GettingStarted/SimpleML-token-reading.cws'
sDesign = 'GettingStarted/SolarSystem0.sml'
messages of parseFreeQuiet(sScript, project, sDesign):
the file has been read successfully
```

See also:

parseAsBNF 4.3.141, parseFree 4.3.142, parseStringAsBNF 4.3.144, translate 4.3.208, translateString 4.3.209, expand 4.3.65, autoexpand 4.3.5, generate 4.3.85, generateString 4.3.86

4.3.145 parseStringAsBNF

- procedure **parseStringAsBNF**(BNFFileName **:** *script*, this **:** *tree*, content **:** *string*)

Parameter	Type	Description
BNFFileName	*script<BNF>*	the name of the *BNF-driven* parsing script
this	*tree*	the current node that will be accessed with this variable
content	*string*	the text to parse

Parses a text, which is given by the argument content as a sequence of characters. It executes the BNF-driven script called BNFFileName; see section 4.3.216 for more information.

See also:

parseAsBNF 4.3.141, parseFree 4.3.142, parseFreeQuiet 4.3.143, translate 4.3.208, translateString 4.3.209, expand 4.3.65, autoexpand 4.3.5, generate 4.3.85, generateString 4.3.86

4.3.146 pathFromPackage

- function **pathFromPackage**(package **:** *string*) **:** *string*

Parameter	Type	Description
package	*string*	a package path

Converts a package path to a directory path. A *package path* is a sequence of identifiers separated by dots. All dots ('**.**') encountered are replaced by a path separator ('\\ under *Windows* and '**/**' on *UNIX* platforms). A path separator is added at the end.

Example:

```
traceLine("pathFromPackage('java.solarsystem') = '" +
pathFromPackage("java.solarsystem") + "'");
```

Output:

```
pathFromPackage('java.solarsystem') = 'java/solarsystem/'
```

4.3.147 postHTTPRequest

- function **postHTTPRequest**(URL **:** *string***,** HTTPSession **:** *treeref***,** arguments **:** *treeref*) **:** *string*

Parameter	Type	Description
URL	*string*	URL of the HTTP server
HTTPSession	*treeref*	an object to describe the HTTP session
arguments	*treeref*	list of the arguments to POST; the key contains the name of the argument and the element gives the value

This function sends an HTTP's POST request to the HTTP server pointed to by the parameter URL with the list of arguments put into the the parameter arguments.

The function returns the document read from the HTTP server.

The function sendHTTPRequest() (see 4.3.179) describes the structure of the HTTP session object.

See also:

getHTTPRequest 4.3.93, sendHTTPRequest 4.3.179

4.3.148 pow

- function **pow**(x **:** *double***,** y **:** *double*) **:** *double*

Parameter	Type	Description
x	*double*	the base
y	*double*	the exponent

Returns value of the argument `x` raised to the power of the second argument `y`. The arguments are converted to numerics, being worth *0* when a conversion fails. The power is then processed, and the result is converted to a string, skipping fractional part if all digits after the dot are *0*.

Example:

```
traceLine("pow(3, 4) = " + pow(3, 4));
```

Output:

```
pow(3, 4) = 81
```

See also:

`add` 4.3.1, `sub` 4.3.197, `mult` 4.3.134, `div` 4.3.47, `exp` 4.3.64, `log` 4.3.130, `mod` 4.3.133

4.3.149 prec

- function **prec(**`i` **:** *iterator***)** **:** *bool*

Parameter	Type	Description
i	*iterator*	iterator of a `foreach` statement or pointing to items of a list

The iterator will now point to the precedent item of the list and returns `true` if exists.

See also:

`first` 4.3.82, `index` 4.3.109, `last` 4.3.122, `key` 4.3.121, `next` 4.3.137, `createIterator` 4.3.38, `createReverseIterator` 4.3.39, `duplicateIterator` 4.3.48

4.3.150 produceHTML

- procedure **produceHTML(**`scriptFileName` **:** *string***,** `HTMLFileName` **:** *string***)**

Parameter	Type	Description
scriptFileName	*string*	a script file of CODEWORKER to highlight
HTMLFileName	*string*	the HTML file that represents the highlighted script

This procedure proposes to highlight a script written for CODEWORKER and to provide the resulting colored script into an HTML file. Only '@' and the text to put into the ouput stream are highlighted.

Example:

```
produceHTML("Scripts/Tutorial/GettingStarted/Tiny-JAVA.cwt",
getWorkingPath() + "Scripts/Tutorial/GettingStarted/Tiny-JAVAhighlight.html"
traceLine("the script file has been highlighted into
'Tiny-JAVAhighlight.html'");
```

Output:

```
the script file has been highlighted into 'Tiny-JAVAhighlight.html'
```

Known bugs:

The procedure needs to be improved, so as to highlight tokens and keywords of the language too. It doesn't work yet on BNF-driven scripts intended to a translation.

4.3.151 putEnv

- procedure **putEnv(**name **:** *string***,** value **:** *string***)**

Parameter	Type	Description
name	*string*	name of the variable environment
value	*string*	new value to assign to the variable environment

If variable name is already part of the environment, its value is replaced by value; otherwise, the new variable and its value are added to the environment. You can remove a variable from the environment by specifying an empty string.

This procedure affects only the environment that is local to the current process; you cannot use them to modify the *command-level* environment. That is, these functions operate only on data structures accessible to the run-time library and not on the environment "segment" created for a process by the operating system. When the current process terminates, the environment reverts to the level of the calling process (in most cases, the operating-system level). However, the modified environment can be passed to any new processes created by the instruction system, and these new processes get any new items added by putEnv.

Example:

```
putEnv("JUST_FOR_FUN", "I'd like to finish reading my
newspaper");
traceLine("getEnv('JUST_FOR_FUN') = '" + getEnv("JUST_FOR_FUN")
+ "'");
```

Output:

```
getEnv('JUST_FOR_FUN') = 'I'd like to finish reading my
newspaper'
```

See also:

getEnv 4.3.91, environTable 4.3.52, existEnv 4.3.60, system 4.3.200

4.3.152 randomInteger

- function **randomInteger() :** *int*

  Generates a pseudorandom number.

  See also:

  randomSeed 4.3.152

4.3.153 randomSeed

- procedure **randomSeed(**seed **:** *int***)**

Parameter	Type	Description
seed	*int*	a new seed for generating pseudorandom integers

Sets the seed for generating a series of pseudorandom integers. To change the seed to a given starting point, choose any positive value different of `1` as the seed argument. A value of `1` reinitializes the generator. Any negative value let CODEWORKER choose a random seed for you.

See also:

`randomInteger` 4.3.151

4.3.154 receiveBinaryFromSocket

- function **receiveBinaryFromSocket(**`socket` **:** *int***,** `length` **:** *int***) :** *string*

Parameter	Type	Description
`socket`	*int*	a client socket descriptor
`length`	*int*	number of bytes to read

This function waits for `length` bytes to read from `socket`, and returns a sequence of bytes (CODEWORKER represents a byte with 2 hexadecimal digits).

If an error occurs, the function returns an empty string.

See also:

`createINETClientSocket` 4.3.36, `createINETServerSocket` 4.3.37, `acceptSocket` 4.3, `attachInputToSocket` 4.5, `detachInputFromSocket` 4.5.3, `attachOutputToSocket` 4.6.3, `detachOutputFromSocket` 4.6.7, `receiveFromSocket` 4.3.154, `receiveTextFromSocket` 4.3.155, `sendTextToSocket` 4.3.180, `sendBinaryToSocket` 4.3.178, `closeSocket` 4.3.18, `flushOutputToSocket` 4.6.10

4.3.155 receiveFromSocket

- function **receiveFromSocket(**`socket` **:** *int***,** `isText` **:** *boolref***) :** *string*

Parameter	Type	Description
`socket`	*int*	a client socket descriptor
`isText`	*boolref*	the function will populate this parameter with `true` if read bytes designate a string and `false` if they are binary data

This function waits for bytes to read from `socket` and returns them. If an error occurs, the function returns an empty string.

The function sets `isText` to:

- `true` if it has received a text,
- `false` if it has received binary data: the returned string is then a sequence of bytes (CODEWORKER represents a byte with 2 hexadecimal digits),

See also:

`createINETClientSocket` 4.3.36, `createINETServerSocket` 4.3.37, `acceptSocket` 4.3, `attachInputToSocket` 4.5, `detachInputFromSocket` 4.5.3, `attachOutputToSocket` 4.6.3, `detachOutputFromSocket` 4.6.7,

```
receiveBinaryFromSocket    4.3.153,    receiveTextFromSocket    4.3.155,
sendTextToSocket 4.3.180, sendBinaryToSocket 4.3.178, closeSocket 4.3.18,
flushOutputToSocket 4.6.10
```

4.3.156   receiveTextFromSocket

- function **receiveTextFromSocket(**`socket` **:** *int***,** `length` **:** *int***) :** *string*

Parameter	Type	Description
socket	*int*	a client socket descriptor
length	*int*	size of the text to read

This function waits for `length` bytes to read from `socket`, and returns a string.

If an error occurs, the function returns an empty string.

See also:

```
createINETClientSocket    4.3.36,    createINETServerSocket    4.3.37,
acceptSocket  4.3,  attachInputToSocket  4.5,  detachInputFromSocket
4.5.3,    attachOutputToSocket    4.6.3,    detachOutputFromSocket    4.6.7,
receiveBinaryFromSocket    4.3.153,    receiveFromSocket    4.3.154,
sendTextToSocket 4.3.180, sendBinaryToSocket 4.3.178, closeSocket 4.3.18,
flushOutputToSocket 4.6.10
```

4.3.157   relativePath

- function **relativePath(**`path` **:** *string***,** `reference` **:** *string***) :** *string*

Parameter	Type	Description
path	*string*	the path to give as relative to `reference`
reference	*string*	a path that serves as the reference to determine the relative path

Returns the relative path that allow going to the well-named `path`, considering the `reference` path as the starting point (like a current directory). Under the *Windows* platform, if the two arguments don't hold on the same drive, the absolute path of the first argument is returned.

Note that the arguments are converted to canonical paths (see `canonicalPath()` `canonicalPath()` for more information).

Example:

```
local sPath = getCurrentDirectory() + "Documentation/CodeWorker.pdf";
traceLine("path = '" + sPath + "'");
local sReference = "WebSite/downloads";
traceLine("reference = '" + sReference + "'");
traceLine("result = '" + relativePath(sPath, sReference) + "'");
```

Output:

```
path = 'E:/projects/generator/Documentation/CodeWorker.pdf'
reference = 'WebSite/downloads'
result = '../../Documentation/CodeWorker.pdf'
```

See also:

4.3.158   removeAllElements

- procedure **removeAllElements**(`variable` : *treeref*)

Parameter	Type	Description
`variable`	*treeref*	an array of nodes

Removes all elements of the array pointed to by `variable`.

Example:

```
local myTree = "monkey";
pushItem myTree["Everest"];
pushItem myTree["Tea spoon"];
traceLine("the array 'myTree' has " + myTree.size() + "
elements");
traceLine("all elements are removed");
removeAllElements(myTree);
traceLine("Is the array 'myTree' empty now?  = '" +
myTree.empty() + "'");
```

Output:

```
the array 'myTree' has 2 elements
all elements are removed
Is the array 'myTree' empty now?  = 'true'
```

See also:

4.3.159   removeDirectory

- function **removeDirectory**(`path` : *string*) : *bool*

Parameter	Type	Description
`path`	*string*	the directory to remove

The function removes the directory specified by `path`. The directory must not be the current working directory or the root directory.

The function returns `false` if the path is invalid or cannot be deleted.

Example:

```
local sDirectory = getWorkingPath() + "Scripts/Tutorial/GettingStarted/bin";
if !removeDirectory(sDirectory) error("impossible to remove '" +
sDirectory + "'");
```

```
changeDirectory 4.3.11, canonizePath 4.3.9, copySmartDirectory 4.3.31,
exploreDirectory 4.3.66, getCurrentDirectory 4.3.90, relativePath 4.3.156,
resolveFilePath 4.3.168, scanDirectories 4.3.175, existDirectory 4.3.59
```

4.3.160   removeElement

- procedure **removeElement**(variable **:** *treeref* **,** key **:** *string*)

Parameter	Type	Description
variable	*treeref*	an array of nodes
key	*string*	the entry key of the element to remove

Removes the element whose entry key is passed to the argument key from the array of nodes called variable.

Example:

```
local myTree = "monkey";
pushItem myTree["Everest"];
pushItem myTree["Tea spoon"];
traceLine("the array 'myTree' has " + myTree.size() + "
elements");
traceLine("element 'Tea spoon' is removed");
removeElement(myTree, "Tea spoon");
traceLine("the array 'myTree' has " + myTree.size() + " elements
now");
```

Output:

```
the array 'myTree' has 2 elements
element 'Tea spoon' is removed
the array 'myTree' has 1 elements now
```

See also:

```
removeAllElements 4.3.157, removeFirstElement 4.3.160, removeLastElement
4.3.162
```

4.3.161   removeFirstElement

- procedure **removeFirstElement**(list **:** *treeref*)

Parameter	Type	Description
list	*treeref*	an array of nodes

Removes the first element from the array of nodes called list.

Nothing occurs if list doesn't exist or is empty.

Example:

```
local myTree = "monkey";
pushItem myTree["Everest"];
```

```
pushItem myTree["Tea spoon"];
traceLine("the array 'myTree' has " + myTree.size() + "
elements");
traceLine("the first element is removed:");
removeFirstElement(myTree);
traceObject(myTree);
```

Output:

```
the array 'myTree' has 2 elements
the first element is removed:
Tracing variable 'myTree':
    "monkey"
    ["Tea spoon"]
End of variable's trace 'myTree'.
```

See also:

`removeAllElements` 4.3.157, `removeElement` 4.3.159, `removeLastElement` 4.3.162

4.3.162 removeGenerationTagsHandler

- function **removeGenerationTagsHandler**(key **:** *string*) **:** *bool*

Parameter	Type	Description
key	*string*	designates the handler to remove

Removes the current generation tags handler amongst those previously registered thanks to the function `addGenerationTagsHandler()`. If the current generation tags handler is worth this one, no custom handler is selected.

Returns `true` if `key` designates a registered handler.

See also:

`addGenerationTagsHandler` 4.3.2, `selectGenerationTagsHandler` 4.3.177

4.3.163 removeLastElement

- procedure **removeLastElement**(list **:** *treeref*)

Parameter	Type	Description
list	*treeref*	an array of nodes

Removes the last element from the array of nodes called `list`.

Nothing occurs if `list` doesn't exist or is empty.

Example:

```
local myTree = "monkey";
pushItem myTree["Everest"];
pushItem myTree["Tea spoon"];
traceLine("the array 'myTree' has " + myTree.size() + "
elements");
```

```
traceLine("the last element is removed:");
removeLastElement(myTree);
traceObject(myTree);
```

Output:

```
the array 'myTree' has 2 elements
the last element is removed:
Tracing variable 'myTree':
    "monkey"
    ["Everest"]
End of variable's trace 'myTree'.
```

See also:

```
removeAllElements 4.3.157, removeElement 4.3.159, removeFirstElement
4.3.160
```

4.3.164   removeRecursive

- procedure **removeRecursive**(variable : *treeref*, attribute : *string*)

Parameter	Type	Description
variable	*treeref*	points to a node of a parse tree
attribute	*string*	the name of an attribute to remove

Removes recursively the attribute called attribute from a parse tree given by variable. It checks also recursively the nodes put into arrays.

Example:

```
local myTree = "to keep";
insert myTree.toKeep = "to keep";
insert myTree.toRemove = "to remove";
insert myTree.toKeep.toRemove = "to remove";
insert myTree.list["keep"].toKeep = "to keep";
insert myTree.list["remove"].toRemove = "to remove";
removeRecursive(myTree, "toRemove");
local theGoal = "to keep";
insert theGoal.toKeep = "to keep";
insert theGoal.list["remove"] = "";
insert theGoal.list["keep"].toKeep = "to keep";
if !equalTrees(myTree, theGoal) error("removeRecursive() doesn't
work!");
traceLine("the attribute 'toRemove' has been removed from
'myTree' recursively");
```

Output:

```
the attribute 'toRemove' has been removed from 'myTree'
recursively
```

4.3.165 removeVariable

- procedure **removeVariable**(node : *treeref*)

Parameter	Type	Description
node	*treeref*	the node to remove from the tree

All attributes of the argument `node` are deleted, its array of nodes is cleared and its value becomes an empty string. If the node was referring to another node, the link is cleared. Once these task are completed, the variable `node` is removed from the tree it belongs to (as an attribute or an element).

Note that trying to remove a local variable throws an error.

Example:

```
local myTree;
insert myTree.nodeToRemove = "the value";
localref myNode = myTree.nodeToRemove;
insert myNode.a1 = "attribute 1";
insert myNode.a2 = "attribute 2";
insert myNode.array["1"] = "node 1";
insert myNode.array["2"] = "node 2";
traceObject(myNode);
traceLine("- the variable 'myNode' is removed:");
removeVariable(myNode);
traceObject(myTree);
```

Output:

```
Tracing variable 'myTree.nodeToRemove':
    "the value"
    a1 = "attribute 1"
    a2 = "attribute 2"
    array
    array["1", "2"]
End of variable's trace 'myTree.nodeToRemove'.
- the variable 'myNode' is removed:
Tracing variable 'myTree':
End of variable's trace 'myTree'.
```

See also:

`existVariable` 4.3.62, `clearVariable` 4.3.17, `findFirstSubstringIntoKeys` 4.3.77, `findElement` 4.3.75, `findNextSubstringIntoKeys` 4.3.80, `getArraySize` 4.3.87, `getVariableAttributes` 4.3.100, `invertArray` 4.3.114, `isEmpty` 4.3.115

4.3.166 repeatString

- function **repeatString**(text : *string*, occurrences : *int*) : *string*

Parameter	Type	Description
text	*string*	the string to repeat
occurrences	*int*	number of times the string must be repeated

Returns the result of repeating the sequence of characters passed to argument `text` a number of times given by the parameter `occurrences`.

Example:

```
traceLine("repeatString('Hungry!', 3') = '" +
repeatString("Hungry!", 3) + "'");
```

Output:

```
repeatString('Hungry!', 3') = 'Hungry!Hungry!Hungry!'
```

See also:

`countStringOccurences` 4.3.34, `completeLeftSpaces` 4.3.22, `completeRightSpaces` 4.3.23, `replaceString` 4.3.166, `replaceTabulations` 4.3.167, `toLowerString` 4.3.201, `toUpperString` 4.3.202, `trimLeft` 4.3.211, `trimRight` 4.3.212, `trim` 4.3.210, `truncateAfterString` 4.3.213, `truncateBeforeString` 4.3.214

4.3.167 replaceString

- function **replaceString**(`old` **:** *string*, `new` **:** *string*, `text` **:** *string*) **:** *string*

Parameter	Type	Description
old	*string*	the substring to be replaced
new	*string*	the string replacing the old one
text	*string*	the sequence of characters to handle

Returns the result of replacing all occurrences of substring passed to argument `old` by the substring `new`, when found into `text`.

Example:

```
local sText = "first in, first out";
traceLine("replaceString('fir', 'la', '" + sText + "') = '" +
replaceString("fir", "la", sText) + "'");
```

Output:

```
replaceString('fir', 'la', 'first in, first out') = 'last in,
last out'
```

Method: *text*.`replaceString`(*old*, *new*)

See also:

`countStringOccurences` 4.3.34, `completeLeftSpaces` 4.3.22, `completeRightSpaces` 4.3.23, `repeatString` 4.3.165, `replaceTabulations` 4.3.167, `toLowerString` 4.3.201, `toUpperString` 4.3.202, `trimLeft` 4.3.211, `trimRight` 4.3.212, `trim` 4.3.210, `truncateAfterString` 4.3.213, `truncateBeforeString` 4.3.214

4.3.168 replaceTabulations

- function **replaceTabulations**(`text` **:** *string*, `tab` **:** *int*) **:** *string*

Parameter	Type	Description
text	*string*	a sequence of characters where spaces must be inserted instead of tabulations
tab	*int*	size of a tabulation

Returns the result of replacing all tabulations (character '**\t**') of the string passed to argument `text` by spaces. The maximum of spaces to insert instead of tabulation is given by the parameter `tab`.

Notice that spaces to insert are determined according to the position of the tabulation in the string and the beginning of the *line* (it means that '**\n**' characters are taken into account) and the tabulation size. So, this function isn't equivalent to `replaceString()` 4.3.166.

Example:

```
local sText = " a little joke";
traceLine("replaceTabulations(sText, 4) = '" +
replaceTabulations(sText, 4) + "'");
traceLine("replaceString('\t', " ", sText) = '" +
replaceString("\t", " ", sText) + "'");
```

Output:

```
replaceTabulations(sText, 4) = ' a little joke'
replaceString(' ', , sText) = ' a little joke'
```

See also:

`countStringOccurences` 4.3.34, `completeLeftSpaces` 4.3.22, `completeRightSpaces` 4.3.23, `repeatString` 4.3.165, `replaceString` 4.3.166, `toLowerString` 4.3.201, `toUpperString` 4.3.202, `trimLeft` 4.3.211, `trimRight` 4.3.212, `trim` 4.3.210, `truncateAfterString` 4.3.213, `truncateBeforeString` 4.3.214

4.3.169   resolveFilePath

- function **resolveFilePath**(`filename` : *string*) : *string*

Parameter	Type	Description
filename	*string*	the path of the file to resolve

Searches the file `filename` in the current directory and, if fails, it continues searching it in the include directories ('`-I`' switch on the command line).

It returns the location of the file in directories, removing any ambiguity.
If the file doesn't exist, the function returns an empty string.
If `filename` points to a virtual file, the function returns `filename`.

Example:

```
local sIncludePath = getIncludePath();
setIncludePath(sIncludePath + ";Documentation");
traceLine("resolveFilePath('CodeWorker.tex') = '" +
resolveFilePath("CodeWorker.tex") + "'");
setIncludePath(sIncludePath);
```

Output:

```
resolveFilePath('CodeWorker.tex') = 'Documentation/CodeWorker.tex'
```

See also:

```
changeDirectory 4.3.11, canonizePath 4.3.9, copySmartDirectory 4.3.31,
exploreDirectory 4.3.66, getCurrentDirectory 4.3.90, relativePath 4.3.156,
removeDirectory 4.3.158, scanDirectories 4.3.175, existDirectory 4.3.59
```

4.3.170   rightString

- function **rightString**(text : *string*, length : *int*) : *string*

Parameter	Type	Description
text	*string*	a sequence of characters
length	*int*	a positive number

Returns the last characters that belong to the string passed to the argument `text`. The number of characters to take is given by argument `length`. If the string contains less than `length` characters, the function returns all of them.

Example:

```
traceLine("rightString('airport', 4) = '" + rightString("airport",
4) + "'");
traceLine("rightString('airport', 8) = '" + rightString("airport",
8) + "'");
```

Output:

```
rightString('airport', 4) = 'port'
rightString('airport', 8) = 'airport'
```

See also:

```
charAt 4.3.13, coreString 4.3.33, cutString 4.3.42, joinStrings 4.3.120,
leftString 4.3.123, lengthString 4.3.124, midString 4.3.132, rsubString
4.3.170, subString 4.3.198
```

4.3.171   rsubString

- function **rsubString**(text : *string*, pos : *int*) : *string*

Parameter	Type	Description
text	*string*	a sequence of characters
pos	*int*	a position starting at 0, and relative to the end of the text string

Returns the sequence of characters passed to argument `text` after skipping the last `pos` characters. It is a *reverse* subString().

Example:

```
local sText = "The lamp of experience";
traceLine("sText = '" + sText + "'");
```

```
traceLine("rsubString(sText, 5) = '" + rsubString(sText, 5) +
"'");
```

Output:

```
sText = 'The lamp of experience'
rsubString(sText, 5) = 'The lamp of exper'
```

See also:

`charAt` 4.3.13, `coreString` 4.3.33, `cutString` 4.3.42, `joinStrings` 4.3.120, `leftString` 4.3.123, `lengthString` 4.3.124, `midString` 4.3.132, `rightString` 4.3.169, `subString` 4.3.198

4.3.172 saveBinaryToFile

- procedure **saveBinaryToFile(**`filename` **:** *string***,** `content` **:** *string***)**

Parameter	Type	Description
filename	*string*	name of the binary file to write into
content	*string*	sequence of bytes (2 hexadecimal digits) to write into the file

Saves the binary `content` to the file `filename`. The parameter `content` concatenates a sequence of hexadecimal digits, so a byte is stored in 2 characters:
binary-content ::= [*byte*]*;
byte ::= ['**0**'..'**9**' | '**A**'..'**F**' | '**a**'..'**f**']2;

The hexadecimal pairs of digit are converted to binary (8 bits) before writing the content.

If the file cannot be created, an error is raised. If the file already exists, its content is replaced by the new binary content.

See also:

`copyFile` 4.3.29, `appendFile` 4.3.4, `changeFileTime` 4.3.12, `chmod` 4.3.16, `copyGenerableFile` 4.3.30, `copySmartFile` 4.3.32, `deleteFile` 4.3.45, `existFile` 4.3.61, `fileCreation` 4.3.69, `fileLastAccess` 4.3.70, `fileLastModification` 4.3.71, `fileLines` 4.3.72, `fileMode` 4.3.73, `fileSize` 4.3.74, `loadBinaryFile` 4.3.126, `loadFile` 4.3.127, `saveToFile` 4.3.174, `scanFiles` 4.3.176

4.3.173 saveProject

- procedure **saveProject(**`XMLorTXTFileName` **:** *string***,** `nodeToSave` **:** *tree***)**

Parameter	Type	Description
XMLorTXTFileName	*string*	an output file that will contain the description of the node to save
nodeToSave	*tree*	**default value:** *project* the node to save; if omitted, it is defaulted to the global variable `project`

Saves the parse tree of the project as a XML or a text file (".xml" or ".txt" extension).

- If XML, each element of the XML hierarchy takes the name of the corresponding attribute in the parse tree. When a value is assigned to an attribute, it is reported into an XML attribute called __VALUE. When an attribute represents an array of nodes, all nodes are inlayed in the body of the XML element like it: each node is put into an XML element called __ARRAY_ENTRY where the XML attribute __KEY contains the entry key.
- if text file, the format is the same as for a constant tree declaration (see section **??**).

Example:

```
parseAsBNF("Scripts/Tutorial/GettingStarted/Tiny-BNFparsing1.cwp",
project, "Scripts/Tutorial/GettingStarted/Tiny.tml");
saveProject(getWorkingPath() +
          "Scripts/Tutorial/GettingStarted/Tiny-tree.xml");
traceLine(loadFile("Scripts/Tutorial/GettingStarted/Tiny-tree.xml"));
```

Output:

```
this file has been parsed successfully
<project>
    <listOfClasses>
        <__ARRAY_ENTRY __KEY="A">
            <name __VALUE="A" />
        </__ARRAY_ENTRY>
        <__ARRAY_ENTRY __KEY="B">
            <name __VALUE="B" />
            <parent __REFERENCE="project.listOfClasses[&quot;A&quot;]"
/>
        </__ARRAY_ENTRY>
        <__ARRAY_ENTRY __KEY="C">
            <name __VALUE="C" />
            <listOfAttributes>
                <__ARRAY_ENTRY __KEY="0">
                    <class __REFERENCE="project.listOfClasses[&quot;B&quot;]"
/>
                    <isArray __VALUE="true" />
                    <name __VALUE="b" />
                </__ARRAY_ENTRY>
            </listOfAttributes>
        </__ARRAY_ENTRY>
        <__ARRAY_ENTRY __KEY="D">
            <name __VALUE="D" />
            <listOfAttributes>
                <__ARRAY_ENTRY __KEY="0">
                    <class __REFERENCE="project.listOfClasses[&quot;A&quot;]"
/>
                    <name __VALUE="a" />
                </__ARRAY_ENTRY>
                <__ARRAY_ENTRY __KEY="1">
                    <class __REFERENCE="project.listOfClasses[&quot;C&quot;]"
/>
                    <isArray __VALUE="true" />
                    <name __VALUE="c" />
                </__ARRAY_ENTRY>
            </listOfAttributes>
```

```
            </__ARRAY_ENTRY>
        </listOfClasses>
</project>
```

See also:

loadProject 4.3.128, saveProjectTypes 4.3.173

4.3.174 saveProjectTypes

- procedure **saveProjectTypes(**XMLFileName **:** *string***)**

Parameter	Type	Description
XMLFileName	*string*	an output file that will contain the XML description of the structure of the main parse tree called project

Factorizes nodes of the parse tree of the project to distinguish an implicit type for nodes, depending on their locations into the graph. The typed tree is saved as an XML file.

Example:

```
parseAsBNF("Scripts/Tutorial/GettingStarted/Tiny-BNFparsing1.cwp",
project, "Scripts/Tutorial/GettingStarted/Tiny.tml");
saveProjectTypes(getWorkingPath() +
            "Scripts/Tutorial/GettingStarted/Tiny-types.xml");
traceLine(loadFile("Scripts/Tutorial/GettingStarted/Tiny-types.xml"));
```

Output:

```
this file has been parsed successfully
<project>
    <listOfClasses>
        <__ARRAY_TYPE name="compulsory" parent="optional[25%]">
            <listOfAttributes>
                <__ARRAY_TYPE class="compulsory"
isArray="optional[66%]" name="compulsory">
                </__ARRAY_TYPE>
            </listOfAttributes>
        </__ARRAY_TYPE>
    </listOfClasses>
</project>
```

Known bugs:

Sometimes, when a type is encountered twice in very different locations of the parse tree, a mistake on the proportion of presence may occur. It will be corrected later.

See also:

loadProject 4.3.128, saveProject 4.3.172

4.3.175 saveToFile

- procedure **saveToFile**(`filename` : *string*, `content` : *string*)

Parameter	Type	Description
`filename`	*string*	name of the text file to write into
`content`	*string*	sequence of characters to write into the file

Saves the text `content` to the file `filename`.

If the file cannot be created, an error is raised. If the file already exists, its content is replaced by the new text content.

See also:

`copyFile` 4.3.29, `appendFile` 4.3.4, `changeFileTime` 4.3.12, `chmod` 4.3.16, `copyGenerableFile` 4.3.30, `copySmartFile` 4.3.32, `deleteFile` 4.3.45, `existFile` 4.3.61, `fileCreation` 4.3.69, `fileLastAccess` 4.3.70, `fileLastModification` 4.3.71, `fileLines` 4.3.72, `fileMode` 4.3.73, `fileSize` 4.3.74, `loadBinaryFile` 4.3.126, `loadFile` 4.3.127, `saveBinaryToFile` 4.3.171, `scanFiles` 4.3.176

4.3.176 scanDirectories

- function **scanDirectories**(`directory` : *tree*, `path` : *string*, `pattern` : *string*) : *bool*

Parameter	Type	Description
`directory`	*tree*	node that will contain the name of filtered files and folders
`path`	*string*	the directory from where to start the exploration
`pattern`	*string*	the filter to apply on files to keep

Explores the directory whose name is passed to the argument `path` and filters all files that validate the `pattern`. The list of files is put into the node's array `directory.`**`files`** and the list of directories are put into the node's array `directory.`**`directories`**. The argument `subfolders` requires exploring sub-directories and each node of the node's array `directory.`**`directories`** repeats the same process recursively. The key of an array's node is the short name of the file or the directory and the value of a directory item is the relative path, whereas the value of a file item is also the short name.

If the directory cannot be found, the variable `directory` doesn't change and the function returns `false`. If the directory doesn't contain any file, the attribute `directory.`*`files`* isn't created. If the directory doesn't contain any subfolder, the attribute `directory.`*`directories`* isn't created.

Example:

```
local theDirectory;
local sPathToExplore = project.winBinaries; // Windows package
of CodeWorker
if !scanDirectories(theDirectory, sPathToExplore, "Leader*.cws")
error("unable to find the directory");
// the complete path is too long:  shorten it
traceLine("starting directory = '" + theDirectory.subString(sPathToExplore.l
+ "':");
foreach j in theDirectory.files {
```

```
        traceLine(" '" + j + "'");
    }
    foreach i in cascading theDirectory.directories {
        // the complete path is too long:  shorten it
        traceLine("- directory '" + i.subString(sPathToExplore.length())
+ "':");
        foreach j in i.directories {
            traceLine(" subfolder '" + key(j) + "'");
        // the complete path is too long:  shorten it
            traceLine(" path '" + j.subString(sPathToExplore.length())
+ "'");
        }
        foreach j in i.files {
            traceLine(" '" + j + "'");
        }
    }
}
```

Output:

```
starting directory = '/':
- directory '/bin/':
- directory '/include/':
- directory '/Scripts/':
    subfolder 'Tutorial'
        path '/Scripts/Tutorial/'
- directory '/Scripts/Tutorial/':
    subfolder 'GettingStarted'
        path '/Scripts/Tutorial/GettingStarted/'
- directory '/Scripts/Tutorial/GettingStarted/':
    'LeaderScript0.cws'
    'LeaderScript1.cws'
    'LeaderScript2.cws'
    'LeaderScript3.cws'
    'LeaderScript4.cws'
    'LeaderScript5.cws'
    'LeaderScript6.cws'
```

See also:

```
changeDirectory 4.3.11, canonizePath 4.3.9, copySmartDirectory 4.3.31,
exploreDirectory 4.3.66, getCurrentDirectory 4.3.90, relativePath 4.3.156,
removeDirectory 4.3.158, resolveFilePath 4.3.168, existDirectory 4.3.59
```

4.3.177   scanFiles

- function **scanFiles**(files : *tree*, path : *string*, pattern : *string*, subfolders : *bool*) : *bool*

Parameter	Type	Description
files	*tree*	node that will contain files that validate the pattern
path	*string*	the directory where to scan files
pattern	*string*	the filter to apply on files to keep
subfolders	*bool*	to scan sub directories recursively

Explores the directory `path` and filters all files that validate the `pattern` given by parameter. Files are put into the node's array called `files` with their relative path, which is assigned to the value of the item. The scan is applied on subfolders if the argument `subfolders` passes `true`.

The `pattern` argument accepts the standard jocker characters ('*' and '?'). If empty, `pattern` is considered as being worth "*".

The function returns `true` if the directory to scan exists.

Example:

```
local files;
local sDirectory = "Scripts/Tutorial/GettingStarted";
if !scanFiles(files, sDirectory, "Leader*.cws", true)
error("impossible to find the directory");
traceLine("filtering recursively all files that conform to
'Leader*.cws'");
foreach i in files {
    traceLine(" " + subString(i, lengthString(sDirectory)));
}
```

Output:

```
filtering recursively all files that conform to 'Leader*.cws'
    /LeaderScript0.cws
    /LeaderScript1.cws
    /LeaderScript2.cws
    /LeaderScript3.cws
    /LeaderScript4.cws
    /LeaderScript5.cws
    /LeaderScript6.cws
```

See also:

`copyFile` 4.3.29, `appendFile` 4.3.4, `changeFileTime` 4.3.12, `chmod` 4.3.16, `copyGenerableFile` 4.3.30, `copySmartFile` 4.3.32, `deleteFile` 4.3.45, `existFile` 4.3.61, `fileCreation` 4.3.69, `fileLastAccess` 4.3.70, `fileLastModification` 4.3.71, `fileLines` 4.3.72, `fileMode` 4.3.73, `fileSize` 4.3.74, `loadBinaryFile` 4.3.126, `loadFile` 4.3.127, `saveBinaryToFile` 4.3.171, `saveToFile` 4.3.174

4.3.178   selectGenerationTagsHandler

- function **selectGenerationTagsHandler**(`key` : *string*) : *bool*

Parameter	Type	Description
key	*string*	designates the handler to take

Selects the current generation tags handler amongst those previously registered thanks to the function `addGenerationTagsHandler()`. If the parameter `key` is worth *false* (empty string), the default generation tags handler is used.

Returns `true` if `key` designates a registered handler.

See also:

`addGenerationTagsHandler` 4.3.2, `removeGenerationTagsHandler` 4.3.161

4.3.179 sendBinaryToSocket

- function **sendBinaryToSocket(**`socket` **:** *int***,** `bytes` **:** *string***) :** *bool*

Parameter	Type	Description
`socket`	*int*	a client socket descriptor
`bytes`	*string*	a sequence of bytes to write

This function writes binary data to a socket and returns `true` if it has achieved successfully.

The function raises an error if a byte passed to `bytes` is malformed: CODEWORKER expects 2 hexadecimal digits to represent a byte.

See also:

`createINETClientSocket` 4.3.36, `createINETServerSocket` 4.3.37, `acceptSocket` 4.3, `attachInputToSocket` 4.5, `detachInputFromSocket` 4.5.3, `attachOutputToSocket` 4.6.3, `detachOutputFromSocket` 4.6.7, `receiveBinaryFromSocket` 4.3.153, `receiveFromSocket` 4.3.154, `receiveTextFromSocket` 4.3.155, `sendTextToSocket` 4.3.180, `closeSocket` 4.3.18, `flushOutputToSocket` 4.6.10

4.3.180 sendHTTPRequest

- function **sendHTTPRequest(**`URL` **:** *string***,** `HTTPSession` **:** *treeref***) :** *string*

Parameter	Type	Description
`URL`	*string*	URL of the HTTP server
`HTTPSession`	*treeref*	an object to describe the HTTP session

This function sends an HTTP request to the HTTP server pointed to by the argument `URL`, and returns the document read from the HTTP server.

If the request fails, an error message is thrown.

The well-named argument `HTTPSession` specifies some information into devoted attributes:

- **agent** *(optional)* is the browser name, "CODEWORKER" by default,
- **referer** *(optional)*,
- **proxy** *(optional)*:
    * **proxy.host** *(compulsory)*,
    * **proxy.port** *(compulsory)*,
    * **proxy.userpwd** *(optional)* is worth "*user***:***password*",
- **cookies** *(optional)* is a list of nodes such as:
    * **name** *(compulsory)*,
    * **value** *(optional)* is worth `""` by default,
    * **path** *(optional)*, populated from the HTTP header (see below)
    * **domain** *(optional)*, populated from the HTTP header (see below)
    * **expires** *(optional)* for a permanent cookie, populated from the HTTP header (see below)

After processing the request successfully, you'll find information about the returned data.

If the data is a binary format, such as an archive or an image, the field `HTTPSession.`**`binary_data`** is worth `true`.

If the data is a textual format, the detail of the received header lines are filled in the array `HTTPSession.`**`header_lines`**. The array `HTTPSession.`**`cookies`** is updated with the cookies extracted from the header. The entry nodes of the array `HTTPSession.`**`header_lines`** are indexed with the name of the header directive. These entry nodes just contain the list of all header values attached to such a directive.

Example:
```
HTTP/1.1 200 OK
Cache-Control:  private
Date:  Wed, 10 Mar 2004 13:41:03 GMT
Server:  Microsoft-IIS/6.0
Set-Cookie:  SESSIONID=Garfield; expires=Wed, 10-Mar-2004
15:03:19 GMT; path=/
Set-Cookie' PREFERENCES=yellow; domain=jupiter; path=/
Content-Type:  text/html
Content-Length:  4469
```

These fields are then injected in the array `HTTPSession.`**`header_lines`**. The header directive `Set-Cookie` appears twice, but gives rise to only one entry node: `HTTPSession.header_lines[`**`"Set-Cookie"`**`]`. This entry node is a list containing two elements. The first one defines all characteristics of `SESSIONID` and the second one provides the characteristics of the cookie `PREFERENCES`.

Here, a piece of code that displays the header lines:
```
foreach i in theSession.header_lines
  foreach j in i
    traceLine("'" + i.key() + "' = '" + j + "'");
```

Output:
```
'HTTP/1.1 200 OK' = ''
'Cache-Control' = 'private'
'Date' = 'Wed, 10 Mar 2004 13:41:03 GMT'
'Server' = 'Microsoft-IIS/6.0'
'Set-Cookie' = 'SESSIONID=Garfield; expires=Wed, 10-Mar-2004
15:03:19 GMT; path=/'
'Set-Cookie' = 'PREFERENCES=yellow; domain=jupiter; path=/'
'Content-Type' = 'text/html'
'Content-Length' = '4469'
```

See also:

`getHTTPRequest` 4.3.93, `postHTTPRequest` 4.3.146

4.3.181   sendTextToSocket

- function **sendTextToSocket(**`socket` **:** *int***,** `text` **:** *string***) :** *bool*

Parameter	Type	Description
`socket`	*int*	a client socket descriptor
`text`	*string*	the text to write

This function writes a text to a socket and returns `true` if it has achieved successfully.

See also:

`createINETClientSocket` 4.3.36, `createINETServerSocket` 4.3.37, `acceptSocket` 4.3, `attachInputToSocket` 4.5, `detachInputFromSocket` 4.5.3, `attachOutputToSocket` 4.6.3, `detachOutputFromSocket` 4.6.7, `receiveBinaryFromSocket` 4.3.153, `receiveFromSocket` 4.3.154, `receiveTextFromSocket` 4.3.155, `sendBinaryToSocket` 4.3.178, `closeSocket` 4.3.18, `flushOutputToSocket` 4.6.10

4.3.182 setCommentBegin

- procedure **setCommentBegin(**`commentBegin` : *string***)**

Parameter	Type	Description
`commentBegin`	*string*	a sequence of characters that represents the beginning of a comment for an output file to handle

Sets the value of a beginning of comment, which is exploited by the procedures taking in charge the source code generation, such as `expand` or `generate`. CODEWORKER must know the format of comments recognized by the output file, to be able to extract or put protected areas, or to detect *expansion markups*. This procedure should be called `before` calling the source code generation, otherwise the new value is ignored by the preprocessing of output files that looks for protected areas and markups.

The beginning of comment assigned by default is worth '//'. This is the symbol of C++ and JAVA comments that are the most frequently files encountered for generation. However, depending on the output file to generate, you'll change the beginning of comment to:

- '/*' to work on a C file,
- '- -' to work on a ADA file,
- '<!- -' to work on a HTML or XML file,
- '%' to work on a LaTeX file,

Note that if the beginning of comments is set to an empty string, protected areas aren't extracted and the expand mode does nothing.

The function `getCommentBegin` allows asking for the last assigned value.

Example:

```
setCommentBegin("<!-");
traceLine("An HTML-XML comment:  '" + getCommentBegin() + "'");
setCommentBegin("%");
traceLine("A LaTeX comment:  '" + getCommentBegin() + "'");
```

Output:

```
An HTML-XML comment:  '<!-'
A LaTeX comment:  '%'
```

See also:

`getCommentBegin` 4.3.88, `getCommentEnd` 4.3.89, `setCommentEnd` 4.3.182

4.3.183 setCommentEnd

- procedure **setCommentEnd(**`commentEnd` **:** *string***)**

Parameter	Type	Description
`commentEnd`	*string*	a sequence of characters that represents the end of a comment for an output file to handle

Sets the value of an end of comment, which is exploited by the procedures taking in charge the source code generation, such as `expand` or `generate`. CODEWORKER must know the format of comments recognized by the output file, to be able to extract or put protected areas, or to detect *expansion markups*. This procedure should be called `before` calling the source code generation, otherwise the new value is ignored by the preprocessing of output files that looks for protected areas and markups.

The end of comment assigned by default is worth '**\r\n**'. This is the symbol of C++ and JAVA comments that are the most frequently files encountered for generation. However, depending on the output file to generate, you'll change the end of comment to:

- '***/**' to work on a C file,
- '**->**' to work on a HTML or XML file,

The function `getCommentEnd` allows asking for the last assigned value.

Example:

```
setCommentEnd("->");
traceLine("An HTML-XML comment ends with:  '" + getCommentEnd()
+ "'");
setCommentEnd("\n");
traceLine("A LaTeX comment ends with:  '" + composeCLikeString(getCommentEnd
+ "'");
```

Output:

```
An HTML-XML comment ends with:  '->'
A LaTeX comment ends with:  '\n'
```

See also:

`getCommentBegin` 4.3.88, `getCommentEnd` 4.3.89, `setCommentBegin` 4.3.181

4.3.184 setGenerationHeader

- procedure **setGenerationHeader(**`comment` **:** *string***)**

Parameter	Type	Description
`comment`	*string*	comment to put into the header

If the text passed to the argument `comment` isn't empty, a comment is added automatically to each file generated with the procedure `generate`. Passing the option `-genheader` on the command line may require the functionality.

This generation header is inlayed in the comment delimiters and conforms to the format:

- – if the comment holds on a single line:
  begin-comment "**##generation header##CodeWorker##**"
      version-number "**##**" *generation-date* "**##**"
      '"' *comment* '"' *end-comment*

- – if the comment holds on more than one line:
  begin-comment "**##generation header##CodeWorker##**"
      version-number "**##**" *generation-date* "**##**" *end-comment*
  begin-comment "**##header start##**" *end-comment*
  begin-comment $line_1$ *end-comment*
  ...
  begin-comment $line_n$ *end-comment*
  begin-comment "**##header end##**" *end-comment*

Changing the generation header doesn't lead to modify the generated file necessary: the header is ignored while comparing two files.

Example:

```
setGenerationHeader("Popeye's Village\nOlive hates spinash");
traceLine("new generation header = '" + getGenerationHeader() +
"'");
local sFileName = "GettingStarted/Tiny-JAVA.cwt";
traceLine("script to execute:");
local sContent = replaceString("\r", "", loadFile(sFileName));
local lines;
cutString(sContent, "\n", lines);
foreach i in lines if !startString(i, "//")
   traceLine("\t" + i);
traceLine("class to generate = '" + project.listOfClasses#[1].name
+ "'");
local sOutputText;
generateString(sFileName, project.listOfClasses#[1],
sOutputText);
traceLine("generated text:");
traceLine(sOutputText);
setGenerationHeader("");
```

Output:

```
new generation header = 'Popeye's Village
Olive hates spinash'
script to execute:
   package tiny;

   public class @
   this.name@ @
   if existVariable(this.parent) {
      @ extends @this.parent.name@ @
   }
   @{
      // attributes:
   @
   function getJAVAType(myAttribute :  node) {
      local sType = myAttribute.class.name;
```

```
        if myAttribute.isArray {
            set sType = "java.util.ArrayList/*<" + sType + ">*/";
        }
        return sType;
    }

    foreach i in this.listOfAttributes {
        @ private @getJAVAType(i)@ _@i.name@ = null;
    @
    }
    @
        //constructor:
        public @this.name@() {
        }

        // accessors:
    @
    foreach i in this.listOfAttributes {
        @ public @getJAVAType(i)@ get@toUpperString(i.name)@() {
return _@i.name@; }
        public void set@toUpperString(i.name)@(@getJAVAType(i)@
@i.name@) { _@i.name@ = @i.name@; }
    @
    }
    setProtectedArea("Methods");
    @}

class to generate = 'Earth'
generated text:
//##generation header##CodeWorker##4.5.1##28jul2008 00:44:34##
//##header start##
//Popeye's Village
//Olive hates spinash
//##header end##
package tiny;

public class Earth extends Planet {
    // attributes:
    private _countryNames = null;

    //constructor:
    public Earth() {
    }

    // accessors:
    public getCOUNTRYNAMES() { return _countryNames; }
    public void setCOUNTRYNAMES( countryNames) { _countryNames =
countryNames; }
//##protect##"Methods"
//##protect##"Methods"
}
```

See also:

`extractGenerationHeader` 4.3.68, `getGenerationHeader` 4.3.92

4.3.185 setIncludePath

- procedure **setIncludePath(**`path` **:** *string***)**

Parameter	Type	Description
`path`	*string*	a concatenation of paths separated by **';'**

It changes the include path passed to the command line with one or more times the setting of the option `-I`.

The include path expects a concatenation of paths separated by semi-commas (**';'**).

Example:

```
local sOldPath = getIncludePath();
setIncludePath("Here/is;better/than;before");
traceLine("one changes the path:  '" + getIncludePath() + "'");
setIncludePath(sOldPath);
```

Output:

```
one changes the path:  'Here/is/;better/than/;before/'
```

See also:

`getProperty` 4.3.97, `getIncludePath` 4.3.94, `getVersion` 4.3.101, `getWorkingPath` 4.3.102, `setProperty` 4.3.186, `setVersion` 4.3.188, `setWorkingPath` 4.3.189

4.3.186 setNow

- procedure **setNow(**`constantDateTime` **:** *string***)**

Parameter	Type	Description
`constantDateTime`	*string*	the current date-time is fixed to this value

Fixes the current date-time to the value passed to argument `constantDateTime`, conforming to the format:
`%d%b%Y %H:%M:%S.%L`

The procedure doesn't change the system time. *now* is just frozen for the scripting language when calling `getNow()`. One passes an empty date-time to unfreeze the time.

For explanations about *format types*, see function `formatDate` at 4.3.84.

Example:

```
// the time is already frozen for building the documentation
local sOldFrozenTime = getNow();
traceLine("now = " + getNow());
traceLine("one freezes the time to '19jan2003 06:30:00.100'");
```

```
setNow("19jan2003 06:30:00.100");
traceLine("now = '" + getNow() + "' is frozen to this value");
setNow(sOldFrozenTime);
```

Output:

```
now = 28jul2008 20:42:00.500
one freezes the time to '19jan2003 06:30:00.100'
now = '19jan2003 06:30:00.100' is frozen to this value
```

See also:

`formatDate` 4.3.84, `addToDate` 4.3.3, `compareDate` 4.3.19, `completeDate` 4.3.21, `getLastDelay` 4.3.95, `getNow` 4.3.96

4.3.187   setProperty

- procedure **setProperty**(`define` : *string*, `value` : *string*)

Parameter	Type	Description
define	*string*	name of a property
value	*string*	value to assign to the property

It assigns the value held by the argument `value` to a property whose name is given by parameter `define`. It is equivalent of writing `'-D define=value'` on the command line.

An error is raised if the `define` argument is an empty string.

Example:

```
setProperty("JUST_FOR_FUN", "Monty Python");
traceLine("getProperty('JUST_FOR_FUN') = '" +
getProperty("JUST_FOR_FUN") + "'");
```

Output:

```
getProperty('JUST_FOR_FUN') = 'Monty Python'
```

Deprecated form: `setDefineTarget` has disappeared since version *1.30*

See also:

`getProperty` 4.3.97, `getIncludePath` 4.3.94, `getVersion` 4.3.101, `getWorkingPath` 4.3.102, `setIncludePath` 4.3.184, `setVersion` 4.3.188, `setWorkingPath` 4.3.189

4.3.188   setTextMode

- procedure **setTextMode**(`textMode` : *string*)

Parameter	Type	Description
textMode	*string*	text mode (binary or not)

Sets the mode of text that must be retained for parsing and source code generation. The argument `textMode` is worth one of the following values:

- **"DOS"**: the default value if the interpreter is running under a *Windows* platform,

---

- **"UNIX"**: the default value if the interpreter isn't running under a *Windows* platform,
- **"BINARY"**: not exploited yet, but intended to specify later that the parsing and the source code generation are applied on binary files,

An exception is raised if the argument `textMode` passes a bad value.

The impact of choosing samp"DOS" instead of any other mode is that special comments, which announce markup keys and protected areas, will finish by "\r\n" when the end of comment is a newline '\n'.

Example:

```
local sTextMode = getTextMode();
traceLine("This documentation is generated under '" + sTextMode
+ "' text mode");
setTextMode("BINARY");
traceLine("Now, it is generated under '" + getTextMode() +
"'!");
setTextMode(sTextMode);
```

Output:

```
This documentation is generated under 'DOS' text mode
Now, it is generated under 'BINARY'!
```

See also:

`getTextMode` 4.3.99

4.3.189 setVersion

- procedure **setVersion**(`version` : *string*)

Parameter	Type	Description
version	*string*	version number of scripts

Indicates to the CODEWORKER interpreter that scripts must be considered as written in an older version of the scripting language, given by the parameter `version`.

It allows CODEWORKER to behave as if it was an ancient interpreter and eventually, to adapt deprecated forms.

Example:

```
local sVersion = getVersion();
traceLine("The version of scripts is '" + sVersion + "'");
setVersion("1.5.2");
traceLine("Now, the version of scripts is '" + getVersion() +
"'");
setVersion(sVersion);
```

Output:

```
The version of scripts is '4.5.1'
Now, the version of scripts is '1.5.2'
```

See also:

`getProperty` 4.3.97, `getIncludePath` 4.3.94, `getVersion` 4.3.101, `getWorkingPath` 4.3.102, `setIncludePath` 4.3.184, `setProperty` 4.3.186, `setWorkingPath` 4.3.189

4.3.190 setWorkingPath

- procedure **setWorkingPath(**`path` **:** *string***)**

Parameter	Type	Description
`path`	*string*	the new working path

Changes the output directory that was assigned to the option **-path** on the command line.

Example:

```
local sOldWorkingPath = getWorkingPath();
setWorkingPath("WebSite/");
traceLine("'old path' = '" + sOldWorkingPath + "'");
traceLine("'new path' = '" + getWorkingPath() + "'");
setWorkingPath(sOldWorkingPath);
```

Output:

```
'old path' = 'e:\Projects\generator/'
'new path' = 'WebSite/'
```

See also:

`getProperty` 4.3.97, `getIncludePath` 4.3.94, `getVersion` 4.3.101, `getWorkingPath` 4.3.102, `setIncludePath` 4.3.184, `setProperty` 4.3.186, `setVersion` 4.3.188

4.3.191 setWriteMode

- procedure **setWriteMode(**`mode` **:** *string***)**

Parameter	Type	Description
`mode`	*string*	is worth `"insert"` or `"overwrite"`

Selects how to write text during a generation and how to apply an implicit copy during a translation.

By default, a text is written in overwrite mode (`mode` = `"overwrite"`): if the file cursor doesn't point to the end of the current output file, the new text overwrites the old one and the remaining, if any, is inserted at the end.
The insert mode (`mode` = `"insert"`) causes a shift of the old text, so as to preserve it.

See also:

`getWriteMode` 4.3.103

4.3.192 shortToBytes

- function **shortToBytes(**`short` **:** *ushort***) :** *string*

Parameter	Type	Description
`short`	*ushort*	an unsigned short integer using the decimal base

---

Converts an unsigned short integer in decimal base to its 2-bytes representation. Bytes are ordered in the host order (memory storage).

Example:

```
traceLine("shortToBytes(255) = '" + shortToBytes(255) + "'");
```

Output:

```
shortToBytes(255) = 'FF00'
```

See also:

byteToChar 4.3.8, bytesToLong 4.3.6, bytesToShort 4.3.7, charToByte 4.3.14, charToInt 4.3.15, hexaToDecimal 4.3.104, longToBytes 4.3.131, octalToDecimal 4.3.139

4.3.193 sleep

- procedure **sleep**(millis : *int*)

Parameter	Type	Description
millis	*int*	how many milliseconds the execution must be suspended

The procedure suspends the execution for millis milliseconds.

4.3.194 slideNodeContent

- procedure **slideNodeContent**(orgNode : *treeref*, destNode : *treexpr*)

Parameter	Type	Description
orgNode	*treeref*	points to a node of a parse tree
destNode	*treexpr*	a branch starting at the orgNode node

Moves the entire content (both attributes and array nodes) of the node passed to the argument orgNode, so as to put it at the extremity of a new branch, added to the original node orgNode once its content has been taken off.

For instance, slideNodeContent(*pExpr, left*) means that the content of *pExpr* slides to *pExpr.left*.

Example:

```
local pExpr;
// a given parsing leads to populate an expression:
insert pExpr.operator = "*";
insert pExpr.left = 3.141592;
insert pExpr.right = "X";
traceLine("'pExpr' represents '3.141592 * X'");
// the parsing continues and reveals that the precedent
// expression was the left part of a bigger arithmetic
// expression:
traceLine("After moving, the content of 'pExpr' becomes the left
hand:");
```

```
slideNodeContent(pExpr, left);
traceLine("' - 'pExpr' contains only the sub-node 'left'");
traceLine("' - 'pExpr.left' represents '3.141592 * X'");
insert pExpr.operator = "+";
insert pExpr.right = "Y";
traceLine("'pExpr' describes now '(3.141592 * X) + Y'");
```

Output:

```
'pExpr' represents '3.141592 * X'
After moving, the content of 'pExpr' becomes the left hand:
'  - 'pExpr' contains only the sub-node 'left'
' - 'pExpr.left' represents '3.141592 * X'
'pExpr' describes now '(3.141592 * X) + Y'
```

See also:

```
equalTrees 4.3.55
```

4.3.195   sortArray

- procedure **sortArray**(array **:** *tree*)

Parameter	Type	Description
array	*tree*	the array node to sort

Sort an array in the lexicographical order of the entry keys.

Example:

```
local myArray;
insert myArray["Garfield"];
insert myArray["Tea spoon"];
insert myArray["Everest"];
traceLine("Sort the array 'myArray':");
sortArray(myArray);
foreach i in myArray {
    traceLine("\t\"" + i.key() + "\"");
}
```

Output:

```
Sort the array 'myArray':
    "Everest"
    "Garfield"
    "Tea spoon"
```

4.3.196   sqrt

- function **sqrt**(x **:** *double*) **:** *double*

Parameter	Type	Description
x	*double*	the number we want to calculate the square root

Calculates the square root of `x`.

An invalid number causes the function to throw an error.

Example:

```
traceLine("sqrt(25) = " + sqrt(25));
```

Output:

```
sqrt(25) = 5
```

4.3.197 startString

- function **startString**(`text` : *string*, `start` : *string*) : *bool*

Parameter	Type	Description
`text`	*string*	a sequence of characters to test
`start`	*string*	the prefix

`"true"` if the argument `start` is a prefix of the character sequence represented by `text`; `""` otherwise. Note also that `"true"` will be returned if `start` is an empty string or is equal to argument `text`.

Example:

```
local sText = "airport";
traceLine("startString('" + sText + "', 'air') = '" +
startString(sText, "air") + "'");
```

Output:

```
startString('airport', 'air') = 'true'
```

See also:

`findFirstChar` 4.3.76, `endString` 4.3.51, `findLastString` 4.3.78, `findNextString` 4.3.79, `findString` 4.3.81

4.3.198 sub

- function **sub**(`left` : *double*, `right` : *double*) : *double*

Parameter	Type	Description
`left`	*double*	first arithmetic member
`right`	*double*	second arithmetic member

Returns the result of arithmetic subtraction `left` - `right`. Members are converted from strings to numbers, supposed being worth 0 if a parsing error occurs; then the subtraction is processed, and the result is converted to a string, skipping fractional part if all digits after the dot are *0*.

Remember that the symbol '**-**' doesn't mean anything in the standard syntax of the language, so there is no way to confuse for expressing a subtraction. However, it exists an escape mode that allows writing arithmetic expressions between '**$**' symbols, as formula under *LaTeX*. So, `$left` `− right$` is equivalent to `sub(left, right)`.

Example:

```
local a = 4.5;
traceLine(a + " - 2.8 = " + sub(a, "2.8"));
traceLine(a + " - 2.5 = " + sub(a, 2.5) + " <- integer value");
```

Output:

```
4.5 - 2.8 = 1.7
4.5 - 2.5 = 2 <- integer value
```

See also:

add 4.3.1, mult 4.3.134, div 4.3.47, exp 4.3.64, log 4.3.130, mod 4.3.133, pow 4.3.147

4.3.199 subString

- function **subString**(text : *string*, pos : *int*) : *string*

Parameter	Type	Description
text	*string*	a sequence of characters
pos	*int*	a position starting at 0, and relative to the beginning of the text string

Returns the sequence of characters passed to argument text after skipping the first pos characters.

Example:

```
local sText = "The lamp of experience";
traceLine("sText = '" + sText + "'");
traceLine("subString(sText, 4) = '" + subString(sText, 4) +
"'");
```

Output:

```
sText = 'The lamp of experience'
subString(sText, 4) = 'lamp of experience'
```

See also:

charAt 4.3.13, coreString 4.3.33, cutString 4.3.42, joinStrings 4.3.120, leftString 4.3.123, lengthString 4.3.124, midString 4.3.132, rightString 4.3.169, rsubString 4.3.170

4.3.200 sup

- function **sup**(left : *double*, right : *double*) : *bool*

Parameter	Type	Description
left	*double*	the first member
right	*double*	the second member

Compares two numbers and returns true if the first member given by argument left is strictly greater than the second member passed to argument right.

Don't use the operator '>' to compare numbers in the classical syntax of the interpreter: it only checks the lexicographical order. So, *'3 > 12'* is true. However, it exists an escape mode that

allows writing arithmetic comparisons between '**$**' symbols, as formula under *LaTeX*. So, `$left > right$` is equivalent to `inf(left, right)`.

Example:

```
traceLine("sup(12, 3) = '" + sup(12, 3) + "'");
traceLine("12 > 3 = '" + (12 > 3) + "'");
```

Output:

```
sup(12, 3) = 'true'
12 > 3 = ''
```

See also:

`equal` 4.3.53, `inf` 4.3.110

4.3.201 system

- function **system**(command **:** *string*) **:** *string*

Parameter	Type	Description
command	*string*	the command interpreter will execute it

This function passes `command` to the command interpreter, which executes the string as an operating-system command. If `command` is empty, the function simply checks to see whether the command interpreter exists and returns an empty string. If an error occurs, it returns the corresponding error message.

Example:

```
local sScript = getWorkingPath() + "Documentation/System.cws";
local sCommand;
if existFile("CodeWorker.exe") set sCommand = "CodeWorker.exe";
else set sCommand = "Release\\CodeWorker";
set sCommand += " -script " + sScript;
local sOutput = getWorkingPath() + "Documentation/System.out";
traceLine("another \"CodeWorker\" is launched from here,");
local sError = system(sCommand + " > " + sOutput);
if sError error(sError);
traceLine("and executes the following script:");
traceLine(loadFile(sScript));
traceLine("the trace was put into a file:");
traceLine(loadFile(sOutput));
```

Output:

```
another "CodeWorker" is launched from here,
and executes the following script:
traceLine("This text comes from an external \"CodeWorker\"");

the trace was put into a file:
CodeWorker v4.5.1 (LGPL), parses and generates source code
easily;
Copyright (C) 1996-2008 Cedric Lemaire; see 'http://www.codeworker.org'.
This text comes from an external "CodeWorker"
```

4.3.202 toLowerString

- function **toLowerString**(`text` **:** *string*) **:** *string*

Parameter	Type	Description
`text`	*string*	string to make in lower case

Converts each uppercase letter in `text` to lowercase, and returns the result. Other characters are not affected

Example:

```
local sText = "BE AFRAID ABOUT the \"WORLD COMPANY\"";
traceLine("toLowerString('" + sText + "') = '" +
toLowerString(sText) + "'");
```

Output:

```
toLowerString('BE AFRAID ABOUT the "WORLD COMPANY"') = 'be
afraid about the "world company"'
```

4.3.203 toUpperString

- function **toUpperString**(`text` **:** *string*) **:** *string*

Parameter	Type	Description
`text`	*string*	string to capitalize

Converts each lowercase letter in `text` to uppercase, and returns the result. Other characters are not affected.

Example:

```
local sText = "THINK different, LEARN about other
civilizations";
traceLine("toUpperString('" + sText + "') = '" +
toUpperString(sText) + "'");
```

Output:

```
toUpperString('THINK different, LEARN about other
civilizations') = 'THINK DIFFERENT, LEARN ABOUT OTHER
CIVILIZATIONS'
```

See also:

countStringOccurences 4.3.34, completeLeftSpaces 4.3.22, completeRightSpaces 4.3.23, repeatString 4.3.165, replaceString 4.3.166, replaceTabulations 4.3.167, toLowerString 4.3.201, trimLeft 4.3.211, trimRight 4.3.212, trim 4.3.210, truncateAfterString 4.3.213, truncateBeforeString 4.3.214

4.3.204   traceEngine

- procedure **traceEngine()**

  Traces some states about the interpreter and the current script.

  See also:

  traceLine 4.3.204, traceObject 4.3.205, traceStack 4.3.206, traceText 4.3.207

4.3.205   traceLine

- procedure **traceLine(**line **:** *string***)**

Parameter	Type	Description
line	*string*	a string expression to display to the console

Evaluates the expression passed to argument line, and displays the resulting string to the console. An end of line is added automatically.

In case of *quiet* execution, there is no output to the console, but the line will be kept:

- for processing if used through the JNI interface, to raise all messages to the JAVA application that exploit the CODEWORKER library,

- for concatenating it into a string that collects all messages and that returns it after calling function executeStringQuiet (see 4.3.58),

Example:

traceLine("A text to display, and then");
traceLine("I go to the next line");

Output:

A text to display, and then
I go to the next line

See also:

traceEngine 4.3.203, traceObject 4.3.205, traceStack 4.3.206, traceText 4.3.207

4.3.206 traceObject

- procedure **traceObject(**object **:** *tree***,** depth **:** *int***)**

Parameter	Type	Description
object	*tree*	a tree node, means any kind of variable
depth	*int*	**default value:** *0*
		display depth of the tree

Displays all sub-nodes (called *attributes*) and the node item's array, if any, belonging to an object passed to argument object.

The value assigned to the object is displayed too, when it isn't an empty string. If an array of nodes exists, then all entry keys are display, followed by the value assigned to the item node if not empty.

Example:

```
local myTree = "monkey";
insert myTree.hobbies = "to eat bretzel";
insert myTree["Everest"] = "mountain";
insert myTree["Tea spoon"] = "silverware";
traceObject(myTree);
```

Output:

```
Tracing variable 'myTree':
    "monkey"
    hobbies = "to eat bretzel"
    ["Everest" -> "mountain", "Tea spoon" -> "silverware"]
End of variable's trace 'myTree'.
```

See also:

traceLine 4.3.204, traceEngine 4.3.203, traceStack 4.3.206, traceText 4.3.207

4.3.207 traceStack

- procedure **traceStack()**

  Displays the stack of local variables recursively.

  In case of *quiet* execution, there is no output to the console, but the entire call stack description will be kept:

  - for processing if used through the JNI interface, raising all messages to the JAVA application that exploit the CODEWORKER library,

  - for concatenating it into a string that collects all messages and that returns it after calling function executeStringQuiet (see 4.3.58),

  Example:

  traceStack();

  Output:

  ```
  ##stack## script:
  ##stack## block:
  ```

```
theDocumentation
theDocumentation:
iNotDocumentedCounter = "0"
iNotDocumentedParameter = "0"
iNoExampleCounter = "44"
iNoSeeAlsoCounter = "21"
iFunctionCounter = "207"
listOfCommands
    listOfCommands["0", "1", "2", "3", "4", "5", "6", "7", "8",
"9"]
##stack## script:
```

See also:

`traceLine` 4.3.204, `traceEngine` 4.3.203, `traceObject` 4.3.205, `traceText` 4.3.207

4.3.208   traceText

- procedure **traceText(**`text` **:** *string***)**

Parameter	Type	Description
`text`	*string*	a string expression to display to the console

Evaluates the expression passed to argument `line`, and displays the resulting string to the console. On the contrary of `traceLine`, there is no carriage return at the end.

In case of *quiet* execution, there is no output to the console, but the text will be kept:

- for processing if used through the JNI interface, to raise all messages to the JAVA application that exploit the CODEWORKER library,

- for concatenating it into a string that collects all messages and that returns it after calling function `executeStringQuiet` (see 4.3.58),

Example:

```
traceText("A text to display, ");
traceText("but I refuse to go to line!");
traceLine("");
```

Output:

```
A text to display, but I refuse to go to line!
```

See also:

`traceLine` 4.3.204, `traceEngine` 4.3.203, `traceObject` 4.3.205, `traceStack` 4.3.206

4.3.209   translate

- procedure **translate(**`patternFileName` **:** *script***,** `this` **:** *tree***,** `inputFileName` **:** *string***,**
  `outputFileName` **:** *string***)**

Parameter	Type	Description
patternFileName	*script<translate>*	file name of the *pattern script*, which merges both the BNF syntax and the source code generation tags
this	*tree*	the current node that will be accessed via *this* variable
inputFileName	*string*	the input file to parse
outputFileName	*string*	the output file to generate

Parses an input file whose name is given by the argument `inputFileName` and generates a translated file given by the argument `outputFileName`, following the instructions of the *pattern script* called `patternFileName`.

The pattern script merges the BNF syntax presented section 4.3.216 with the source code generation syntax described section 4.5.35.

See also:

`expand` 4.3.65, `autoexpand` 4.3.5, `generate` 4.3.85, `generateString` 4.3.86, `parseAsBNF` 4.3.141, `parseFree` 4.3.142, `parseFreeQuiet` 4.3.143, `parseStringAsBNF` 4.3.144, `translateString` 4.3.209

4.3.210   translateString

- function **translateString**(`patternFileName` **:** *script***,** `this` **:** *tree***,** `inputString` **:** *string*)
 : *string*

Parameter	Type	Description
patternFileName	*script<translate>*	file name of the *pattern script*, which merges both the BNF syntax and the source code generation tags
this	*tree*	the current node that will be accessed via *this* variable
inputString	*string*	the input string to parse

Parses the input string given by the argument `inputString` and returns the output string resulting of the translation, following the instructions of the *pattern script* called `patternFileName`.

The pattern script merges the BNF syntax presented section 4.3.216 with the source code generation syntax described section 4.5.35.

See also:

`parseAsBNF` 4.3.141, `parseFree` 4.3.142, `parseFreeQuiet` 4.3.143, `parseStringAsBNF` 4.3.144, `translate` 4.3.208, `expand` 4.3.65, `autoexpand` 4.3.5, `generate` 4.3.85, `generateString` 4.3.86

4.3.211   trim

- function **trim**(`string` **:** *stringref*) **:** *int*

Parameter	Type	Description
string	*stringref*	variable that contains the characters to be trimmed

This function trims heading and trailing whitespace characters from the referenced argument `string` and returns the number of characters that were removed. It removes newline, space, and tab characters.

Example:

```
local sText = " Spaces for sale!  ";
traceLine("trim(sText) = '" + trim(sText) + "'");
```

Output:

```
trim(sText) = '8'
```

Deprecated form: `trimString` has disappeared since version *1.40*

See also:

```
countStringOccurences    4.3.34,    completeLeftSpaces    4.3.22,
completeRightSpaces 4.3.23, repeatString 4.3.165, replaceString 4.3.166,
replaceTabulations   4.3.167,   toLowerString   4.3.201,   toUpperString
4.3.202, trimLeft 4.3.211, trimRight 4.3.212, truncateAfterString 4.3.213,
truncateBeforeString 4.3.214
```

4.3.212 trimLeft

- function **trimLeft(**`string` **:** *stringref***) :** *int*

Parameter	Type	Description
string	*stringref*	variable that contains the characters to be trimmed

This function trims leading whitespace characters from the argument `string` and returns the number of characters that were removed. It removes newline, space, and tab characters.

Example:

```
local sText = " Spaces for sale!  ";
traceLine("trimLeft(sText) = '" + trimLeft(sText) + "'");
```

Output:

```
trimLeft(sText) = '4'
```

Deprecated form: `trimLeftString` has disappeared since version *1.40*

See also:

```
countStringOccurences    4.3.34,    completeLeftSpaces    4.3.22,
completeRightSpaces 4.3.23, repeatString 4.3.165, replaceString 4.3.166,
replaceTabulations   4.3.167,   toLowerString   4.3.201,   toUpperString
4.3.202,  trimRight  4.3.212,  trim  4.3.210,  truncateAfterString  4.3.213,
truncateBeforeString 4.3.214
```

4.3.213 trimRight

- function **trimRight(**`string` **:** *stringref***) :** *int*

Parameter	Type	Description
string	*stringref*	variable that contains the characters to be trimmed

This function trims trailing whitespace characters from the referenced argument `string` and returns the number of characters that were removed. It removes newline, space, and tab characters.

Example:

```
local sText = " Spaces for sale!  ";
traceLine("trimRight(sText) = '" + trimRight(sText) + "'");
```

Output:

```
trimRight(sText) = '4'
```

Deprecated form: `trimRightString` has disappeared since version *1.40*

See also:

`countStringOccurences` 4.3.34, `completeLeftSpaces` 4.3.22, `completeRightSpaces` 4.3.23, `repeatString` 4.3.165, `replaceString` 4.3.166, `replaceTabulations` 4.3.167, `toLowerString` 4.3.201, `toUpperString` 4.3.202, `trimLeft` 4.3.211, `trim` 4.3.210, `truncateAfterString` 4.3.213, `truncateBeforeString` 4.3.214

4.3.214 truncateAfterString

- function **truncateAfterString**(`variable` : *treeref*, `text` : *string*) : *string*

Parameter	Type	Description
`variable`	*treeref*	this parameter passes a string to handle and receives its truncation
`text`	*string*	a sequence of characters to find into the value held by `variable`

This function:

- searches the sequence of characters passed to the argument `text` into the string given by the parameter `variable`,
- assigns the substring standing on the right of `text` (so, `text` is excluded) to the parameter `variable`,
- returns the left part of the substring that hasn't been assigned.

If the sequence of characters `text` isn't found into the value of `variable`, the function returns an empty string, and the variable doesn't change.

Example:

```
local sVariable = "my tongue fell out with my teeth";
traceLine("sentence = '" + sVariable + "'");
local sResult = truncateAfterString(sVariable, "out");
traceLine("Now, the variable is worth '" + sVariable + "'");
traceLine("And the result = '" + sResult + "'");
```

Output:

```
sentence = 'my tongue fell out with my teeth'
Now, the variable is worth ' with my teeth'
And the result = 'my tongue fell out'
```

See also:

4.3.215 truncateBeforeString

- function **truncateBeforeString**(`variable` **:** *treeref* **,** `text` **:** *string*) **:** *string*

Parameter	Type	Description
`variable`	*treeref*	this parameter passes a string to handle and receives its truncation
`text`	*string*	a sequence of characters to find into the value held by `variable`

This function:

- searches the sequence of characters passed to the argument `text` into the string given by the parameter `variable`,

- assigns the substring standing on the left of `text` (so, `text` is excluded) to the parameter `variable`,

- returns the right part of the substring that hasn't been assigned.

If the sequence of characters `text` isn't found into the value of `variable`, the function returns an empty string, and the variable doesn't change.

Example:

```
local sVariable = "my tongue fell out with my teeth";
traceLine("sentence = '" + sVariable + "'");
local sResult = truncateBeforeString(sVariable, "out");
traceLine("Now, the variable is worth '" + sVariable + "'");
traceLine("And the result = '" + sResult + "'");
```

Output:

```
sentence = 'my tongue fell out with my teeth'
Now, the variable is worth 'my tongue fell '
And the result = 'out with my teeth'
```

See also:

4.3.216 UUID

- function **UUID**() **:** *string*

This function generates an UUID and returns it as a string. An UUID is a 128-bit universally unique id, used by Microsoft and being proposed as an internet standard.

Example:

```
traceLine("generation of an UUID = '" + UUID() + "'");
traceLine("generation of another one = '" + UUID() + "'");
```
Output:
```
generation of an UUID = '120e8620-0c66-4808-9a48-4122d403a8ba'
generation of another one = '02159b61-5f8c-4a1b-be12-a886464f94e1'
```

4.4   The extended BNF syntax for parsing

A BNF description of a grammar is more flexible and more synthetic than a procedural description of parsing. CODEWORKER accepts parsing scripts that conform to a BNF.

BNF is the acronym of *Backus-Naur Form*, and consists of describing a grammar with production rules. The first production rule that is encountered into the script and that isn't a special one (beginning with a '#' like the #empty clause), is chosen as the main non-terminal to match with the input stream, when the *BNF-driven* script is executed.

A non-terminal (often called a *clause* in the documentation) breaks down into terminals and other non-terminals. Defining how to break down a non-terminal is called a production rule. A clause is valid as soon as the production rule matches its part of the input stream.

The syntax of a clause looks like:
```
["#overload"]?  <clause_specifier> <preprocessing> "::=" <sequence>
['|' <sequence>]* ';'
```

where:
```
<preprocessing> ::= "#!ignore" | "#ignore" ['(' <ignore-mode> ')']?
';'
<ignore-mode> ::= "blanks" | "C++" | "JAVA" | "HTML" | "LaTeX";
<sequence> ::= non-terminal | terminal; <terminal> ::= symbol of the
language:  a constant character or string
```

A sequence is a set of terminals and non-terminals that must match the input stream, starting at the current position. A production rule may propose alternatives: if a sequence doesn't match, the engine tries the next one (the alternation symbol '|' separates the sequences).

A regular expression asks for reading tokens into the input stream. If tokens are put in sequence, one behind the other, they are evaluated from the left to the right and all of them must match the input stream. For example, `"class" '{'` is a sequence of 2 non-terminals, which requires that the input stream first matches with *"class"* and then is followed by *'{'*.

Putting **#overload** just before the declaration of a production rule means that the non-terminal was already defined and that it must be replaced by this new rule when called. Example:
```
nonterminal ::= "bye";
...
#overload nonterminal ::= "bye" | "quit" | "exit";
```

Now, calling *nonterminal* executes the second production rule. Use the directive **#super** to call the overloaded clause. The precedent overloading might be written:
```
...
#overload nonterminal ::= #super::nonterminal | "quit" | "exit";
```

`#overload` takes an important place in the reuse of BNF scripts. A parser might be built as reusing a scanner, where some non-terminals only have to be extended, for populating a parse tree for instance.

The statement **`#transformRules`** provides also a convenient way to reuse a BNF script.

It defines a rule that describes how to transform the header (left member) and the production rule (right member) of a non-terminal declaration.

Example:
`INTEGER ::= #ignore ['0'..'9']*;`

INTEGER is the header and *#ignore ['0'..'9']** is the production rule.

During the compilation of a BNF parse script, before processing the declaration of a non-terminal, the compiler checks whether a transforming rule validates the name of the non-terminal. If so, both the header of the declaration and the production rule are translated, following the directives of the rule.

The *#transformRules* statement must be put in the BNF script, before the production rules to transform.

The syntax the statement **`#transformRules`** looks like:
```
transform-rules ::= "#transformRules" filter header-transformation
prod-rule-transformation
filter ::= expression
header-transformation ::= '' translation-script ''
prod-rule-transformation ::= '' translation-script ''
```

The *filter* is a boolean expression, applied on the name of the production rule. The variable **x** contains the name of the production rule.

header-transformation consists on a translation script, which describes how to transform the header. If the block remains empty, the header doesn't change.

prod-rule-transformation consists on a translation script, which describes how to transform the production rule. If the block remains empty, the header doesn't change.

Example:

This example describes how to transform each production rule, whose name ends with `"expr"`.

```
or_expr ::= and_expr ["&&" and_expr]*;
```
becomes
```
or_expr(myExpr :  node) ::= and_expr(myExpr.left)
["&&":myExpr.operator and_expr(myExpr.right)]*;
```

The original production rules are just scanning the input, and the example shows how to transform them for populating a node of the parse tree.

```
 #transformRules
    // The filter accepts production rules that have a name
    // ending with "expr" only.
    // Note that the variable x holds the name
    // of the production rule.
    x.endString("expr")


A script for transforming the header of the production rule:
    {
        // By default, copies the input to the output
        #implicitCopy
        // Writes the declaration of the parameter myExpr
```

```
        // after the non-terminal and copies the rest.
        header ::= #readIdentifier
            => {@(myExpr :  node)@}
            ->#empty;
    }
```

A script for transforming the production rule itself:
```
    {
        #implicitCopy
        // - Pass the left member of the expression to populate,
        // to the first non-terminal,
        // - assign the operator to the expression,
        // - Pass the right member of the expression to populate,
        // to the first non-terminal.
        // In any case, the rest of the production rule remains
        // invariant.
        prodrule ::= [
                #readIdentifier
                =>{@(myExpr.left)@}
                ->[
                   "'" #readChar "'" => {@:myExpr.operator@}
                  |
                     #readCString => {@:myExpr.operator@}
                ]
                #readIdentifier
                =>{@(myExpr.right)@}
            ]?
            ]->#empty;
    }
```

4.4.1 BNF tokens

Below are described all BNF tokens that CODEWORKER recognizes:

- a *constant string*:

Heading	Description
Syntax	A C-like string, written between double quotes and that admits escape sequences: `"C-like_string"`
Matching	The token `"C-like_string"` is valid if it matches the sequence of characters that starts at the current position of the input stream. And then, the position moves just after the string.
Procedural way	This BNF token is equivalent to: `readIfEqualTo("C-like_string")`
Example	`"bottle"` means that the sequence of 6 characters that starts at the current position of the input stream must match with *bottle*.

- a *constant character*:

Heading	Description
Syntax	A C-like character, written between single quotes and that admits escape sequences: `'C-like_char'`
Matching	The token `"C-like_char"` is valid if it matches the character that stands at the current position of the input stream. And then, the position moves to the next character.
Procedural way	This BNF token is equivalent to: `readIfEqualTo("C-like_char")`
Example	`'('` means that the current character of the input stream must be an opening parenthesis.

- an *range of characters*:

Heading	Description
Syntax	Two C-like characters, written between single quotes, which admit escape sequences, and separated by **..**: `'lower_char_boundary'..'upper_char_boundary'`
Matching	The token `'lower_char_boundary'..'upper_char_boundary'` is valid if the character that stands at the current position of the input stream is comprise between the two boundaries (included). And then, the position moves to the next character. Note that `'lower_char_boundary'` must be smaller than `'upper_char_boundary'` considering the ASCII order.

Procedural way	This BNF token is equivalent to the boolean expression:
	`(peekChar() >= "lower_char_boundary") &&`
	`(peekChar() <= "upper_char_boundary") && readChar()`

Example	`'a'..'z'` means that the current character of the input stream must be a lower case letter with no accent.

- the *complementary* of an expression:

Heading	Description
Syntax	An expression preceded by the symbol ˆor ˜: `ˆexpression` or `~expression`
Matching	The token is valid when the expression failed to match the input stream at the current position. And then, position of the input stream moves to the next character.

Procedural way	This BNF token is equivalent to the function:
	<pre>function validateThisSpecialComplementary() {
	    local iLocation = getInputLocation();
	    if /*evaluate the expression*/ {
	        setInputLocation(iLocation);
	        return false;
	    } else readChar();
	    return true;
	}</pre>

Example	`~"bottle"` means that the 6 next characters of the input stream must be different from *bottle*. If so, the position of the input stream moves to the next character.

- the *negation* of an expression:

Heading	Description
Syntax	An expression preceded by the symbol '!': `!expression`
Matching	The token is valid when the expression failed to match the input stream at the current position. On the contrary of the complementary, the position of the input stream **doesn't move** to the next character.

Procedural way	This BNF token is equivalent to the function:
	<pre>function validateThisSpecialNegation() {
	    local iLocation = getInputLocation();
	    if /*evaluate the expression*/ {
	        setInputLocation(iLocation);
	        return false;
	    }
	    return true;
	}</pre>

Example	`!"bottle"` means that the 6 next characters of the input stream must be different from *bottle*.

- reading a character:

Heading	Description
Syntax	Keyword `readChar` preceded by the symbol **'#'**: `#readChar`
Matching	If the end of the input stream hasn't been reached yet, the token is valid and the position of the input stream points to the next character.
Procedural way	This BNF token is equivalent to: `readChar()`
Example	`#readChar` means that the position points to the next character if the end of the input stream hasn't been reached yet.

- reading a byte:

Heading	Description
Syntax	Keyword `readByte` preceded by the symbol **'#'**: `#readByte`
Matching	If the end of the input stream hasn't been reached yet, the token is valid and the position of the input stream points to the next character/byte. It scans a byte, which is converted to a 2-hexadecimal digits.
Procedural way	This BNF token is equivalent to: `readByte()`
Example	`#readByte:sByte` means that the position points to the next character if the end of the input stream hasn't been reached yet and, if the byte *0x0F* was at the current position, the variable *sByte* is worth *"0F"*.

- reading a sequence of bytes:

Heading	Description
Syntax	Keyword `readBytes` preceded by the symbol **'#'**, and followed by an expression, whose result is a positive integer that specifies how many bytes have to be scanned: `#readBytes(`*integer-expression*`)`
Matching	If the end of the input stream hasn't been reached yet, the token is valid and the position of the input stream points after the next *integer-expression* bytes. It scans N bytes (N resulting of the evaluation of *integer-expression*), which are converted to a sequence of 2-hexadecimal digits.
Procedural way	This BNF token is equivalent to: `readBytes(`N`)`
Example	`#readBytes(`*4*`):sBytes` means that the position points to the next four bytes if the end of the input stream hasn't been reached yet. If the sequence of bytes *0x0F0E0D0C* was at the current position, the variable *sBytes* is worth *"0F0E0D0C"*.

- reading a sequence of characters:

Heading	Description
Syntax	Keyword `readChars` preceded by the symbol `'#'`, and followed by an expression, whose result is a positive integer that specifies how many characters have to be scanned: `#readChars(`*integer-expression*`)`
Matching	If the end of the input stream hasn't been reached yet, the token is valid and the position of the input stream points after the next *integer-expression* characters. It scans `N` characters, `N` resulting of the evaluation of *integer-expression*).
Procedural way	This BNF token is equivalent to: `readChars(`*N*`)`
Example	`#readChars(`*4*`):`*sChars* means that the position points to the next four characters if the end of the input stream hasn't been reached yet. If the sequence of characters *"cats and dogs"* was at the current position, the variable *sChars* is worth *"cats"*.

- reading a C-like constant character:

Heading	Description
Syntax	Keyword `readCChar` preceded by the symbol `'#'`: `#readCChar`
Matching	The non-terminal BNF symbol `#readCChar` reads a C-like constant character. In case of assignement of the scanned value to a variable, only the constant character is returned, without the single quotes. A C-like character stands between single quotes and admits the escape character \.
Example	`#readCChar:`*sValue1* `#readCChar:`*sValue2* applied to *'A'*'\n' will move the position of the input stream after the second trailing single quote. The variable *sValue1* will contain the letter `A` and *sValue2* will contain the *newline* characters.

- reading a C-like string:

Heading	Description
Syntax	Keyword `readCString` preceded by the symbol `'#'`: `#readCString`
Matching	The token is valid if the character that stands at the current position of the input stream is a double quote (`"`) and that if all following characters up to the trailing double quote are recognized as C-like characters (meaning also that the escape sequences are as presented for the function `composeCLikeString()` 4.3.25).
Procedural way	This BNF token is equivalent to: `readString(`*sString*`)`
Example	`#readCString:`*sValue* applied to *"Popeye's village\nOlive hates spinash"* will move the position of the input stream after the trailing double quote. The variable *sValue* will contain the string without double quotes and such as the escape sequences are converted to their ASCII representation.

- reading an identifier:

Heading	Description
Syntax	Keyword `readIdentifier` preceded by the symbol **'#'**: `#readIdentifier`
Matching	The token is valid if the characters that stand starting at the current position of the input stream match the following BNF sequence: `#!ignore ['a'..'z'|'A'..'Z'|'_']` `['a'..'z'|'A'..'Z'|'_'|'0'..'9']*`**//**
Procedural way	This BNF token is equivalent to: `readIdentifier()`
Example	`#readIdentifier` applied to *mushroom12,carpet* will move the position of the input stream on the comma after reading *mushroom12*.

- reading insignificant characters:

Heading	Description
Syntax	**"#skipIgnore"** [`'('` *<ignore-mode>* `')'`]? where *<ignore-mode>* specifies what is an insignificant character (see `#ignore`).
Matching	It reads insignificant characters, such as blanks and comments, as specified via the BNF directive `#ignore(...)`. this BNF directive is always valid, even if the parsing has already reached the end of file or if there are no insignificant characters.**//**
Procedural way	This BNF token is equivalent to: `skipIgnore()`
Example	`#skipIgnore` applied to */*mushroom12,carpet*/.fish* will move the position of the input stream on the dot before *fish* (if `#ignore(C++) was set before, for instance`).

- reading up to the first insignificant characters:

Heading	Description
Syntax	**"#readUptoIgnore"** [`'('` *<ignore-mode>* `')'`]? where *<ignore-mode>* specifies what is an insignificant character (see `#ignore`).
Matching	It reads all significant characters, up to encountering a whitespace or a comment, which style was specified via the BNF directive `#ignore(...)`. this BNF directive is valid if it reads at least one significant character.**//**
Example	`#readUptoIgnore` `#readUptoIgnore` applied to *mushroom12/*comment*/carpet fish* will read *"mushroom12"* (first directive) and *"carpet"* (second call) before moving the position of the input stream on the whitespace that stands before *fish* (if `#ignore(C++) was set before, for instance`).

- reading an integer:

Heading	Description
Syntax	Keyword `readInteger` preceded by the symbol **'#'**: `#readInteger`
Matching	The token is valid if the characters that stand starting at the current position of the input stream match the following BNF sequence: `#!ignore ['-']?  ['0'..'9']+`**//**
Procedural way	This BNF token is equivalent to: `readInteger(sInteger)`
Example	**#**`readInteger` applied to *12.34* will move the position of the input stream on the dot after reading *12*.

- reading a numeric:

Heading	Description
Syntax	Keyword `readNumeric` preceded by the symbol **'#'**: `#readNumeric`
Matching	The token is valid if the characters that stand starting at the current position of the input stream match the following BNF sequence: `#!ignore [#readInteger ['.'  ['0'..'9']*]?  |` `'.' ['0'..'9']*] [['e'	'E'] ['+'	'-']?`   `['0'..'9']+]?`**//**
Procedural way	This BNF token is equivalent to: `readNumber(sNumber)`
Example	**#**`readNumeric` applied to *12.34E+3mushrooms* will move the position of the input stream at the beginning of *mushrooms*.

- matching the result of a classical string expression:

Heading	Description
Syntax	Keyword `readText` preceded by the symbol **'#'**, and followed by the classical string expression to match, between parenthesis: **#**`readText(`*classical-expression*`)`
Matching	The token is valid if the classical expression matches the input stream at the current position. If so, the position moves just after the expression that matched.
Procedural way	This BNF token is equivalent to: `readIfEqualTo(`*classical-expression*`)`
Example	**#**`readText(`sName`)` where *sName = "horse"* means that the next 5 characters encountered to the input stream at the current position must match *"horse"*, which is the result of the evaluation of the expression.

Classical string expression is seen in opposite of *expressions of tokens*, as being a classical expression of the scripting language (concatenation, call of functions, reference to variables, ...) and not a combination of tokens.

- checking the validity of a classical string expression:

Heading	Description
Syntax	Keyword `check` preceded by the symbol **'#'**, and followed by the classical string expression to check, between parenthesis: `#check(`*classical-expression*`)`
Matching	The token is valid if the classical expression returns a populated string. The position of the input stream is never impacted by this clause.

Procedural way	This BNF token is equivalent to:
	`if `*classical-expression*` // call the rest of the`
	`sequence`

Example	`#check(`*sID* `== "`*struct*`")` `'{'` `[attribute]*` `'}'` means that if *sID* is equal to *"struct"*, we try to match the reading of attributes embedded between braces. The checking has no effect on the current position of the input stream.

Classical string expression is seen in opposite of *expressions of tokens*, as being a classical expression of the scripting language (concatenation, call of functions, reference to variables, ...) and not a combination of tokens.

- checking whether the end of the stream has been reached or not:

Heading	Description
Syntax	Keyword `empty` preceded by the symbol **'#'**: `#empty`
Matching	The token is valid if the position of the input stream points to the end.

Procedural way	This BNF token is equivalent to the function:
	`function validateEndOfFile() {`
	`    if readChar() {`
	`        goBack();`
	`        return false;`
	`    }`
	`    return true;`
	`}`

Example	`#empty` is valid if the end of the input stream has been reached.

- repeating an expression:

Heading	Description
Syntax/matching	The expression is put between brackets and followed by symbols that determine the multiplicity:
	– **[**`expression`**]**: the expression must match the input stream once (and perhaps more, but it isn't checked); in fact, it is used to impose the priority of token evaluation (for adding the '**|**' operator to ask for a *'OR'*),
	– **[**`expression`**]?**: the expression may be absent or present once (and perhaps more, but it isn't checked); this token is always `true`,
	– **[**`expression`**]***: the expression may be absent or matches as long as possible; this token is always `true`,
	– **[**`expression`**]+**: the expression must match at least once and as long as possible; it is equivalent to the sequence **[**`expression`**] [**`expression`**]***
	– **[**`expression`**]**`iterations`: the expression must be repeated *iterations* times (and perhaps more, but it isn't checked),
	– **[**`expression`**]**`begin..end`: the expression must be repeated a number of times included in the range [*begin*, *end*] (and perhaps more than *end*, but it isn't checked); boundaries are constant integers,
	– **[**`expression`**]**`begin..`*****: the expression must be repeated at least *begin* times and as long as possible, *begin* being a constant integer; note that the star might be replaced by **'n'**,
	– **[**`expression`**]#repeat(**`begin-expr`**)**: the expression must be repeated *begin-expr* times exactly; *begin-expr* is an arithmetic expression evaluated just before processing the regular expression, and returning a positive integer,
	– **[**`expression`**]#repeat(**`begin-expr, end-expr`**)**: the expression must be repeated at least *begin-expr* times but not more than *end-expr* times (could be more, but not checked); *begin-expr* and *end-expr* are arithmetic expressions evaluated just before processing the regular expression; they both return a positive integer,
Procedural way	This BNF token is equivalent to do a loop, controlling the iterations according to the required multiplicity, and returning `true` or `false`.

- finding the next occurrence of a given BNF expression:

Common syntax: **->**`BNF-expression`

Matching: The token is valid if the *BNF expression* is found somewhere into the input stream. The, the cursor jumps to the end of the sub sentence scanned by the *BNF expression*.

The BNF operator '**->**' admits a syntax extension, for the adjustment of its internal mechanism.

->*A* jumps just after the the first matching of *A* in the sentence. It processes the equivalent piece of *extended-BNF* script:
```
=> local iLocation;
~[A => iLocation = getInputLocation();]*
```

```
=> setInputLocation(iLocation);
```

To intervene on the boundaries of the repeated sequence **[A ...]***, an extension was added to the syntax: **—>***boundaries* A, where *boundaries* gives the multiplicity of the bracketed sequence ('?','+','2..*', #repeat(iBegin, iEnd) ...).
Note: the boundaries must be declared just after the arrow.

The text covered by **—>***A* includes the unmatched characters plus the sub sentence scanned by *A*, so **—>***A*:*v* assigns the complete covered text to the variable *v*. This is sometimes a drawback: perhaps do you want to take the unmatched character or the sub sentence scanned by *A*.
So now, you can specify the variables intended to receive these intermediate values:
—>(:*varBefore* **—:***varAfter***)***A*:*varTotal*
Note: the intermediate variables are declared just before the BNF symbol *A*, after the boundaries if any.

Example:
We will apply **[—>(:***varBefore* **—:***varAfter***)** #readNumeric**]:***varTotal* on the sentence **"**Garfield.laziness 99.99 percent**"**:

– *varBefore* = "Garfield.laziness "

– *varAfter* = "99.99"

– *varTotal* = "Garfield.laziness 99.99"

The last extension brought to the jump operator **—>***A* is to allow the execution of a BNF sequence at the beginning of the sub sequence matched by *A*. This BNF sequence is declared into the parenthesis used for the intermediate variables, behind these variables, if any:
—>(:*varBefore* **—:***varAfter* ***B*)***A*

The advantage of infiltrating the BNF sequence *B* is that the intermediate variables are populated, and that the cursor doesn't point after the matching of *A* yet, but at the beginning of the sub sentence matched by *A*.

Procedural way: This BNF token is equivalent to the function:
```
function validateFindToken() {
    local bSuccess;
    do {
        set bSuccess = // expression is expanded here;
      } while !bSuccess && readChar();
    return bSuccess;
}
```
Example: —>"C**"** matches the stream *"ABCD"* and the cursor in the input stream points to *'D'*.

- repeats the execution of a BNF sequence by iterating all items of an array.

  The BNF representation of this statement is:
  ```
  BNF_foreach_statement ::= "#foreach" '(' iterator "in"
  [direction]?  ')'  BNF-sequence
                  [sorted_declaration]?  [cascading_declaration]?
  list-node body_statement
  direction ::= "reverse"
  sorted_declaration ::= "sorted" ["no_case"]?  ["by_value"]?
  cascading_declaration ::= "cascading" ["first" | "last"]?
  ```

  A *foreach* BNF directive iterates all items of the list owned by node *list-node*. The *iterator* refers to the current item of the list, and the following BNF sequence is executed on it.

- restricting the scan to a *sub sentence*.

The sequence `A |> B` is understood as considering the *sub sentence* scanned by `A`, which delimits the portion of text left visible to `B`. `B` starts scanning at the beginning of the *sub sentence* covered by `A` and cannot go beyond.

Once the operator has achieved with success, the cursor points to the end of the *sub sentence* covered by `A`.

You'll find below the different steps processed by the operator:

- the BNF literal `A` is executed,

- if success, the cursor comes back to the beginning of the sub sentence covered by `A`

- then, `B` is executed, knowing that the operator forces the end of the *sub sentence* where `A` had finished,

- if success, the cursor points to the end of `A`, even if `B` hadn't scan up to the end of the sentence.

Example:

We want to recognize all colons present on a line. The *sub sentence* we would like to scan is a line: `->'\n'`. Recognizing colons is like: `[->':']*`, which asks for jumping from a colon to another, without considering the end of line.
`->'\n' |> [->':']*` restricts the colon recognition to the line.

- calling a clause to match its rules:

Heading	Description
Syntax	The name of the clause to match and, if any, the expected parameters between parenthesis separated by commas: `clause-name` or `clause-name(P1, ...Pn)`
Matching	The token is valid if the clause matches at the current position of the input stream.

Procedural way	This BNF token is equivalent to the function:
	`function validateClause() {`
	`    return /*matching of rules of the clause*/;`
	`}`

Example	`INTEGER '.' INTEGER` with `INTEGER ::= ['0'..'9']+` means that the clause must match at least one digit into the input stream, and be followed by a dot and then by another positive integer.

- offering a choice between 2 sequences of BNF tokens:

Heading	Description
Syntax	The 2 expressions are separated by the symbol 'l': *sequence1* **|** *sequence2*
Matching	The token is valid if the clause matches one of the two sequences at the current position of the input stream.
Procedural way	This BNF token is equivalent to: `/*evaluate sequence1*/ || /*evaluate sequence2*/`
Example	**"class"** *IDENT* **'{'** | **"interface"** *IDENT* **';'** with *IDENT* `::= ['a'..'z' | 'A'..'Z']+` means that the clause must match at least one digit into the input stream, and be followed by a dot and then by another positive integer.

4.4.2   Preprocessing of a clause

If no processing has been specified to a clause, characters will be ignored into the input stream, following the instruction of the *ignore* mode (determined by the predefined clause `#ignore`), just before running the clause.

Sometimes, it arrives that the *ignore* mode should change before calling the clause. Let's imagine that C++ comments and blanks are ignored, except at some places where a line-comment is expected, holding a description. If the clause that matches the line-comment is called `description`, each time a description has to be read, the following sequence must be written:
```
#ignore(blanks) description:sDescription #ignore(C++)
```

Thanks to the preprocessing of clause, it is possible to require a specific *ignore* mode while calling a clause. For example:
```
description #ignore(blanks) ::= "//" #!ignore [~['\r' |
'\n']]*:description;
```

On our example, each time a description has to be read, calling the clause *description* is naturally reduced to:
```
description:sDescription
```

4.4.3   Inserting instructions of the scripting language

Instruction of the scripting language may be inserted into a sequence of tokens, and are considered as valid, except when the controlling sequence is interrupted by the `break` statement. These instructions doesn't apply a matching on the input stream, but they serve generally to check the consistence of data and to populate the parse tree. They are announced by the symbol **'=>'**:
```
"=>" instruction ';' or
```
```
"=>" compound-statement where a compound-statement is a block of instructions between
```
braces.

Example:
```
class_declaration(myClass :  node) ::=
    "class" IDENT:myClass.name
    => traceLine("name = '" + myClass.name + "'");
    [
      ':'   IDENT:sParent
      => {
```

```
        if !findElement(sParent, listOfClasses)
          error("class '" + sParent + "' hasn't been declared yet!");
        ref myClass.parent = listOfClasses[sParent];
      }
    ]?
    '{' ...
```

The first swapping to the scripting language is just an instruction to trace, which must end with a semi-colon and that isn't the end of the clause! The second swapping to the script language implements a little more work that is put between braces.

Be careful about declaration of local variables. If you declare a variable into a compound statement, it disappears once the controlling sequence leaves the scope. To declare a variable local to the clause, you can do:

```
...
=> local myVariable;
```

In some particular cases, you may have to execute a BNF sequence from within from such a piece of common script. The only way is to use the directive `#applyBNFRule` followed by a non-terminal call.

4.4.4 Common properties of BNF tokens

The sequence of characters that a BNF token has matched may be assigned to a variable. Then the variable may follow the token, separated by a colon:
token ':' variable_name

Example:

`IDENT : sName` (where *IDENT* `::= ['a'..'z' | 'A'..'Z']+`) means that if the clause *IDENT* is valid, the identifier matching the BNF token is assigned to *sName*. Be careful that if the variable doesn't exist, it is pushed into the stack, on the contrary of a classic CODEWORKER script that asks for declaring explicitly a local variable.

You can also specify to concatenate the text covered by the BNF token, to the ancient value of the variable:
B:**+***v*.

Example:
If *v* is worth n̈ebula:¨ and if the sentence starts with *Örion.*¨, then *v* becomes n̈ebula:Orion¨ after the resolution of:
`#readIdentifier`:**+***v*

The sequence of characters that a BNF token has matched may be worth a constant or may belong to a set of values. Then, the constant or the set of values is following the token, separated by a colon, as for variables:
token ':' constant_value [*':' variable_name*] or
token ':' '{' values_of_the_set '}'':' variable_name where
values_of_the_set ::= constant_value [*',' constant_value*]*

Examples:

  - `IDENT : "class", "interface"` (where *IDENT* `::= ['a'..'z' | 'A'..'Z']+`) means that the identifier must be worth `"class"` or `"interface"`. It isn't equivalent to `["class" | "interface"]`, because this new clause matches the beginning of `"class`*if*`y"` or `"interface`*s*`"` and that's not what is expected.

- #readString : "tea spoon", "fork" : *sSilverware* means that the string must be worth "tea spoon" or "fork" and that the parsed value will be assigned to the variable called *sSilverware*.

4.4.5 BNF directives

Some directives are available:

- **#**appendedFile(*classical-expression*): this directive is valid only for *pattern* and *translation* scripts. The directive is put into a sequence of tokens and will be applied on the rest of the sequence: the *classical-expression* is evaluated and gives the name of an output file, which becomes the new output stream where code generation will be processed, such as text is appended to the output file.

- **#**break: this directive is put in a repeat token, ending a sequence of tokens. It leaves successfully the closer *repeat token* into which the directive was put.

  Example:
  `["--" #break | attribute]2,*`

  Here, the engine reads at least one attribute, and leaves with success when it encounters the string "--". To succeed, the #break interruption cannot occur before the second iteration of this BNF sequence. Otherwise, the minus boundary isn't reached.

  Note that brackets with a multiplicity of **1** (brackets used as parenthesis for changing the priority of *BNF sequence/alternation* resolutions) propagate the **#break** interruption to the closer *repeat token* it is inlayed in.

  Sometimes, the action of leaving a repeat token depends on a condition. A conditional break looks like:
  `#break "(" conditional-expression ")"` **Example:**

  ```
  [
  => local bCond;
  ['a' | 'b':bCond]
  #break(bCond) // force leaving only once 'b' is consumed
  ]*
  ```

- **#**continue: this directive is put into a sequence of tokens, and means that the rest of the token's sequence must match with the input stream. If not, an error is raised, giving the call stack and the position of the input stream where the mistake has occurred.

  Example:
  The following sentence is passed to the interpreter for scanning:
  ```
  a -> b // we have forgotten the semi-comma!
  b -> c;
  ```
  something that looks like state transitions for an automaton.

  Now, we write the corresponding BNF-parsing script:
  ```
  TRANSITIONS ::= [TRANSITION]* #empty => traceLine("OK!");;
  TRANSITION ::= STATE "->" STATE ';';
  STATE ::= #readIdentifier;
  ```
  You notice that the syntax reclaims a semi-comma to close a transition. So, the sentence shows

a syntax error at the first line. The BNF script will fail applying the non-terminal *TRANSITION*, so the scanning will stop before running the `traceLine()` procedure. It fails silently: the head of the grammar doesn't match the sentence and that's all. The interpreter cannot guess that the failure is due to a semi-comma: the non-terminal *TRANSITION* doesn't match, that's right, but perhaps that the caller (the non-terminal *TRANSITIONS* here) proposes an alternative?

To constraint the interpreter to raise a syntax error automatically, you have to employ the `#continue` keyword:

TRANSITION `::=` *STATE* `"->"` `#continue` *STATE* `';';`

means that once the antecedent of a state transistion is detected, the rest of the production rule must be valid in any case. If the sequence following `#continue` (*STATE* `';'` here) fails, a detailed syntax error is thrown. Considering our sentence, the BNF script will raise a syntax error about the lacking semi-comma at the first line.

Not using the directive `#continue` obliges you to write something like:

TRANSITION `::=` *STATE* `"->"` `/*syntax checking made by hand from here*/`
       `[`*STATE*` | => `**`error(`**`"syntax error:  STATE token expected"`**`);`**`]`
       `[';' | => `**`error(`**`"syntax error:  ';' expected"`**`);`**`];`

However, the BNF engine composes automatically the syntax in a way that may not be very expressive on a end-user point of view. Notably, it displays the BNF literal that failed. Hopefully, you can specify a custom error message, available for the BNF sequence following the `#continue` directive:

`#continue` **`("`** *syntax-error-variable* **`","`** *custom-error-message* **`)"`**

- *syntax-error-variable*: name of a previously declared variable, assigned with the default error message that the BNF engine had originally composed;

- *custom-error-message*: expression allowing the concatenation of previouly declared variables; this expression now defines the format of the error message.

Example:

```
=> local sWord;
#continue(syntaxError, "unknown instruction '" + sWord + "' encountered;\nthe
[
#readIdentifier:sWord
instruction<sWord>
]*
```

- **#**`explicitCopy`: this directive is available in a *source-to-source translation* script and may be put:

  - inside a sequence of tokens, meaning that the text scanned by the rest of the sequence won't be copied into the output stream automatically,

  - outside the clauses, at the same level as their declaration, meaning that the entire input stream will be parsed without copying the scanned text into the output stream ; this is the default behavior,

If source code has to be put into the output stream, the developer must specify it explicitly between `'@'` symbols in a compound statement announced by **=>**. See `#implicitCopy` to put the scanned text to the output stream automatically.

- `#generatedFile(`*`classical-expression`*`)`: this directive is valid only for *pattern* and *translation* scripts. The directive is put into a sequence of tokens and will be applied on the rest of the sequence: the *classical-expression* is evaluated and gives the name of an output file, which becomes the new output stream where code generation will be processed. It allows changing the output file during the translation, to split it into a few files for example.

- `#generatedString(`*`variable`*`)`: this directive is valid only for *template-based* and *translation* scripts. The directive is put into a sequence of tokens and will be applied on the rest of the sequence: the output file is redirected into the string variable passed to the argument *variable*, which becomes the new output stream where code generation will be processed. If you don't care about the result, you can pass `null` instead of a string variable.

- `#ignore`: this directive is put into a sequence of tokens, and means that the rest of the token's sequence must ignore blanks and comments before evaluating tokens. It exists some different formats available:

  - `#ignore`: it calls a clause implemented by the user, which is also named `#ignore` (see section 4.4.6),
  - `#ignore(`*`"constant-string"`*`)`: it also calls a clause implemented by the user, but known as attached to a specific identifier. This custom ignore clause is named `#ignore[`*`"constant-string"`*`]` (see section 4.4.6),
  - `#ignore(blanks)`: it ignores blank characters (spaces, new lines, tabulations, carriage returns, ...), considered as having an ASCII code smaller than 32 but not null,
  - `#ignore(spaces)`: it ignores spaces and tabulations only,
  - `#ignore(C++)`: it ignores blank characters and C comments ('**/* ... */**') and line comments ('**// ...**' up to the end of the line),
  - `#ignore(HTML)`: it ignores blank characters and SGML comments ('**<!- - ... - ->**'),
  - `#ignore(JAVA)`: it ignores blank characters and C comments ('**/* ... */**') and line comments ('**// ...**' up to the end of the line),
  - `#ignore(LaTeX)`: it ignores line comments that are start with the '**%**' character, but not spaces or empty lines that have a signification,

- `#!ignore`: this directive is put into a sequence of tokens, and means that the rest of the token's sequence will not ignore blanks or comments between tokens. It works recursively when evaluating a clause call. It is useful for reading a number literal for instance, where digits must be glued together.

- `#implicitCopy`: this directive is available in a *source-to-source translation* script and may be put:

  - inside a sequence of tokens, meaning that the text scanned by the rest of the sequence is copied into the output stream automatically,
  - outside the clauses, at the same level as their declaration, meaning that the entire input stream will be parsed and the scanned text will be copied into the output stream,

  `#implicitCopy` means that the scanned text is copied to the output stream as long as the pattern matching succeeds. If a rule fails, the scanned text is removed from the output stream, up to the last valid token. This isn't the default behavior of a *translation* mode. See `#explicitCopy` to switch this mode off.

- `#insert(`*`variable`*`)` *`sequence-of-BNF-instructions`*: the directive creates a new node *variable* (if it doesn't exist yet) and executes the sequence of BNF instructions that follow.

If the sequence fails and if the node was created by the directive, the *variable* is removed.

This directive is very useful while populating the parse tree, when some choices are proposed: either populate this node or another, or a branch...

To write:

```
#insert(myNode) tryNewNode(myNode)
```

is equivalent to:

```
=> local bCreated = !existVariable(myNode);
=> if bCreated insert myNode;
[
        tryNewItem(myNode)
    |
        => if bCreated removeVariable(myNode);
        #check(false)
]
```

- **#**matching(*variable*): this directive is put outside the production rules. It informs the BNF engine to record the coverage of the input text by the production rules. Concretely, the BNF engine stores into a variable the list of all production rules of the grammar and all areas they match in the input text, once the execution has finished. The BNF engine populates the variable specified by the directive.

```
  // file "Documentation/MatchingStructure.txt":
Tree structure of the variable populated by the BNF engine
for #matching(variable):

* variable
  |
  +- rules[]:  list of production rules (signatures only),
  |
  +- areas[]:  table of positions in the input text, the
    | key is worth the position P; no item value
   +- begin[]:  (optional) table for every rule starting
   | | at the position P, the key being worth the
   | | ending position Pf (decreasing order)
   | +- []:  list of all rules matching [P, Pf] exactly,
   |
   +- end[]:  (facultative) table for every rule ending at
      | the position P, the key being worth the
      | starting position Pi (increasing order)
     +- []:  list of all rules matching [Pi, P] exactly,
```

- **#**moveAhead: Located into a BNF sequence, it means that after a valid matching of the rest of the sequence, at least one character must have been consumed.

Example:
```
#moveAhead [A]? [B]?
```

If A doesn't match the input file, then B must match so that the scan has read at least one character.

- **#**nextStep: this directive is put into BNF sequences inlayed in a BNF jump operator (**->**) or in a BNF complementary operator (` or ˜). Normally, these operators move the cursor of the input stream one position further, in case of failure while applying the BNF sequence.

  Using **#nextStep** allows changing the shift of the cursor to more than one character. It is very useful when you encounter to quoted strings or identifiers. For instance, if you are looking for constant strings and then number or identifiers, the following code is incorrect:
  ->[#readCString #readInteger **|** #readIdentifier**]**

  Why is it incorrect? If the constant string matches but not the integer, the next iteration will put the cursor just after the quote and will perhaps point to an identifier, embedded into the constant string. Then, the operator will match during the second iteration.

  In fact, if the constant string matches but not the rest of the BNF sequence, we want to force the next jump just after the constant string:
  ->[#readCString **#nextStep** #readInteger **|** #readIdentifier**]**

- **#**noCase: this directive is put:

  - inside a sequence of tokens, meaning that the rest of the sequence must match without taking into account the case of the letters,

  - outside the clauses, at the same level of their declaration, meaning that the entire input stream will be parsed without taking into account the case of letters,

- **#**parsedFile(*filename*): this directive is valid only for *parsing* and *translation* scripts. The directive is put into a sequence of tokens and will be applied on the rest of the sequence: the input file is redirected into the file passed to the argument *filename*, which becomes the new input stream where scanning and parsing will be processed.

- **#**parsedString(*expression*): this directive is valid only for *parsing* and *translation* scripts. The directive is put into a sequence of tokens and will be applied on the rest of the sequence: the input stream is redirected to the text resulting of the evaluation of the argument *expression*, which becomes the new input stream where scanning and parsing will be processed.

- **#**pushItem(*variable*) *sequence-of-BNF-instructions*: the directive pushes a new item into the array of the node *variable* and executes the sequence of BNF instructions that follow. If the sequence fails, the last element of the array *variable* is removed.
  This directive is very useful while populating the parse tree, when some choices are proposed: either populate this array or another, or a branch...
  To write:

  ```
  [#pushItem(list) tryNewItem(list#last)]+
  ```

  is equivalent to:

```
                => local bCreated = !existVariable(list);
                [
                        [
                                => pushItem(list);
                                tryNewItem(list#last)
                        ]+
                    |
                        => {
                                if bCreated removeVariable(list);
                                else removeLastElement(list);
                        }
                ]
```

- **#**`ratchet`: When encountered in a production rule, the BNF engine memorizes what is the current position in the input stream, and then controls that the scan will never come back before this position.

- **#**`super` **"::"** *clause_name*: this directive applies to a non-terminal call, which was overloaded via the #`overload` keyword. The directive means that the underlying non-terminal must be called in place of the overloadee clause. If the non-terminal wasn't overloaded, an error is thrown.

- **#**`trace`: this directive traces the resolution steps of the grammar. Hit a key to interrupt the display of trace information and to pause the controlling sequence.

- **#**`try` *sequence-of-BNF-instructions* **#**`catch`**(**`variable`**)** *sequence-of-BNF-instructions*: The *try/catch* statement catches all error messages thrown from the embedded sequence. The error message is available in the variable passed to the *catch* statement. If no error occurs, the flow of control continues on the sequence following the *catch*. In case of error raising, the sequence breaks at the *catch statement* in failure after populating the variable with the error message.
  Example:

```
#try
non_terminal_call
#catch(sError)
=> traceLine("No trouble! There was no error thrown.");
|
=> traceLine("Error! Message = '" + sError + "'");
```

4.4.6 Declaring a clause

We have seen that a clause may expect some arguments. Such a kind of clause conforms to the syntax:
clause_specifier ::= clause_name [parameters]? [':' return_type]?
'::=' clause_body
clause_name ::= identifier [template_resolution]?;
template_resolution ::= '<' [identifier | constant_string] '>';
parameters ::= '(' parameter [',' parameter]* ')'
parameter ::= argument_name ':' argument_mode
argument_mode ::= **"value"** *|* [**"node"** *|* **"variable"**] *|* **"reference"**

```
return_type ::= "list" | "node" | "value"
clause_body ::= rule_expression ';'
```

where the *argument mode* means:

Mode	Description
value	the parameter is passed by value to the clause, as for user-defined functions
node or variable	the parameter expects a tree node, as for user-defined functions
reference	the parameter expects a reference to a variable, which allows changing the node pointed to by the variable, as for user-defined functions

Example:

```
attribute_declaration(myAttribute :  node, sClassName :  value) ::=
type_specifier(myAttribute.type) IDENT:myAttribute.name;
```

While reusing production rules from a scanner to build a parser, for example, the non-terminal symbols of the parser need to pass a node intended to be fulfilled with parsing information, or to contain some data about the context.

It exists a special clause the user may have to define, named **#**ignore. It allows the implementation of its own production rule for processing empty characters between tokens.

This clause doesn't expect any parameter:
#ignore ::= ...   /*the production rule of how to skip characters*/;

To activate it in a production rule, type **#ignore** with no parameter.

In some cases, you might have to define more than one customized *#ignore* clause. It is possible too, assigning a key to each new special clause while their implementation:
#ignore["the key"**]** ::= ...   /*the production rule of how to skip characters*/;

To activate it in a production rule, type **#ignore(**"the key"**)** with no parameter, as you could have written #ignore(C++) for activating a predefined *ignore mode*.

Note that these special clauses must figure at the beginning of the *extended-BNF* script, before the first appearance for activation in a production rule.

4.5   Reading tokens for parsing

The functions and procedures described below are available in a kind of parsing scripts: those which read tokens in a procedural way, proposing a set of appropriate functions and procedures. All examples that illustrate how to exploit them are applied on the floowing text to parse:

```
    // file "Documentation/ParsingSample.txt":
    identifier:  _potatoes41$
    numbers:  42 23.45e6
    string:  "a C-like string that accepts backslash-escape
sequences"
    word:  1\'ecurie_1stable
    blanks:
        "blanks are ignored"
    spaces:  "spaces are ignored"
    C++:  /*comment*/
```

```
      // other comment
      "blanks and C++ comments are ignored"
   HTML: <!-comment->
      "blanks and HTML comments are ignored"
   LaTeX: % comment
   "blanks must be skipped explicitly"
      "only comments were ignored"
```

There is no syntax extension provided for this mode of parsing, so it is really considered as *procedure-driven*, in the opposite of the *BNF-driven* mode that has been seen in the precedent section.

4.5.1   attachInputToSocket

- procedure **attachInputToSocket(**`socket` **:** *int***)**

Parameter	Type	Description
`socket`	*int*	a client socket descriptor

Joins the input stream of a parsing script to a socket stream: each time that the input stream pointer reaches the end, the interpreter waits for bytes coming from the socket.

Waiting for bytes is a blocking process, so once you don't expect for other bytes anymore, don't forget to detach the socket via the procedure `detachInputFromSocket()` before reaching the end of the stream.

See also:

`detachInputFromSocket` 4.5.3, `createINETClientSocket` 4.3.36, `createINETServerSocket` 4.3.37, `acceptSocket` 4.3, `attachOutputToSocket` 4.6.3, `detachOutputFromSocket` 4.6.7, `receiveBinaryFromSocket` 4.3.153, `receiveFromSocket` 4.3.154, `receiveTextFromSocket` 4.3.155, `sendTextToSocket` 4.3.180, `sendBinaryToSocket` 4.3.178, `closeSocket` 4.3.18, `flushOutputToSocket` 4.6.10

4.5.2   countInputCols

- function **countInputCols() :** *int*

Determines the column number in the line where the parse cursor points to.

See also:

`countInputLines` 4.5.2

4.5.3   countInputLines

- function **countInputLines() :** *int*

Determines the current line number where the parse cursor points to.

See also:

`countInputCols` 4.5.1

4.5.4   detachInputFromSocket

- procedure **detachInputFromSocket**(socket **:** *int*)

Parameter	Type	Description
socket	*int*	a client socket descriptor

Disconnects the input stream of a parsing script from a socket stream. You should have join the socket to the input stream before, via the procedure `attachInputToSocket()`.

See also:

`createINETClientSocket` 4.3.36, `createINETServerSocket` 4.3.37, `acceptSocket` 4.3, `attachInputToSocket` 4.5, `attachOutputToSocket` 4.6.3, `detachOutputFromSocket` 4.6.7, `receiveBinaryFromSocket` 4.3.153, `receiveFromSocket` 4.3.154, `receiveTextFromSocket` 4.3.155, `sendTextToSocket` 4.3.180, `sendBinaryToSocket` 4.3.178, `closeSocket` 4.3.18, `flushOutputToSocket` 4.6.10

4.5.5   getInputFilename

- function **getInputFilename() :** *string*

Returns the path of the input file being parsed.

4.5.6   getInputLocation

- function **getInputLocation() :** *int*

Returns the current file position for reading the input stream.

Example:

```
// we move further into the input file, just after the '$'
character
readNextText("$");
traceLine("The character following '$' is put at position " +
getInputLocation() + ", starting at 0");
```

Output:

```
The character following '$' is put at position 24, starting at 0
```

Deprecated form: `getLocation` has disappeared since version *3.7.1*

See also:

`setInputLocation` 4.5.28, `goBack` 4.5.7

4.5.7   getLastReadChars

- function **getLastReadChars**(length **:** *int*) **:** *string*

Parameter	Type	Description
length	*int*	number of characters to read

Returns the last `length` characters that have been read: it takes up to the number of characters passed to the argument `length`, characters that are preceding the current position of the input file.

Example:

```
// we move further into the input file
readNextText("$");
traceLine("getLastReadChars(12) = '" + getLastReadChars(12) +
"'");
```

Output:

```
getLastReadChars(12) = '_potatoes41$'
```

Deprecated form: `readLastChars` has disappeared since version *1.30*

See also:

`peekChar` 4.5.9, `readByte` 4.5.11, `readBytes` 4.5.12, `readChar` 4.5.14, `readChars` 4.5.16, `readCharAsInt` 4.5.15, `readIdentifier` 4.5.17, `readLine` 4.5.21, `readAdaString` 4.5.10, `readNumber` 4.5.23, `readPythonString` 4.5.24, `readString` 4.5.25, `readWord` 4.5.27

4.5.8 goBack

- procedure **goBack()**

  Moves back the position of the input stream, pointing to the character just before. If the current position was pointing to the beginning of the input stream, the function has no effect.

  Example:

  ```
  traceLine("we move further into the input file, just after
  '$'");
  readNextText("$");
  traceLine("and now, we go back to it");
  goBack();
  if !readIfEqualTo("$") error("'$' expected");
  ```

  Output:

  ```
  we move further into the input file, just after '$'
  and now, we go back to it
  ```

  See also:

  `setInputLocation` 4.5.28, `getInputLocation` 4.5.5

4.5.9 lookAhead

- function **lookAhead(**`text` **:** *string***) :** *bool*

Parameter	Type	Description
text	*string*	a sequence of characters to match

Checks whether the next characters of the input stream match with the string passed to argument `text`. If so, the function returns `true` and the position of the input stream hasn't moved.

Example:

```
// we move further into the input file,
// just after 'C++:  '
readNextText("C++:  ");
local iPosition = getInputLocation();
traceLine("lookAhead('/*') = '" + lookAhead("/*") + "'");
if iPosition != getInputLocation() error("What did I say?  The
file position shouldn't have moved!");
```

Output:

```
lookAhead('/*') = 'true'
```

See also:

`readIfEqualTo`   4.5.18,   `readIfEqualToIgnoreCase`   4.5.20,
`readIfEqualToIdentifier` 4.5.19

4.5.10   peekChar

- function **peekChar()** : *string*

  Returns the character found at the current input stream position, or an empty string if the end of file has been reached. If succeeded, the position of the input file **doesn't move** to the next character.

  Example:

  ```
  setInputLocation(10);
  traceLine("at position 10, peekChar() = '" + peekChar() + "'");
  if !equal(getInputLocation(), 10) error("the position of the
  input stream shouldn't have moved!");
  traceLine("the position didn't change, peekChar() = '" +
  peekChar() + "' again");
  ```

  Output:

  ```
  at position 10, peekChar() = ':'
  the position didn't change, peekChar() = ':'  again
  ```

  See also:

  `getLastReadChars` 4.5.6, `readByte` 4.5.11, `readBytes` 4.5.12, `readChar` 4.5.14,
  `readChars` 4.5.16, `readCharAsInt` 4.5.15, `readIdentifier` 4.5.17, `readLine` 4.5.21,
  `readAdaString` 4.5.10, `readNumber` 4.5.23, `readPythonString` 4.5.24, `readString`
  4.5.25, `readWord` 4.5.27

4.5.11   readAdaString

- function **readAdaString**(`text` : *stringref*) : *bool*

Parameter	Type	Description
`text`	*stringref*	a variable that will contain the string literal extracted from the input stream

Reads a string literal surrounded by double quotes, and where the double-quote character has to be repeated.

If succeeded, the position moves just after the trailing double quote and the function returns `true`.

See also:

`peekChar` 4.5.9, `getLastReadChars` 4.5.6, `readByte` 4.5.11, `readBytes` 4.5.12, `readChar` 4.5.14, `readChars` 4.5.16, `readCharAsInt` 4.5.15, `readIdentifier` 4.5.17, `readLine` 4.5.21, `readNumber` 4.5.23, `readPythonString` 4.5.24, `readString` 4.5.25, `readWord` 4.5.27

4.5.12 readByte

- function **readByte()** : *string*

Returns the byte read at the current file position, or an empty string if the end of file has been reached. If succeeded, the position of the input file moves to the next character.

The byte is returned as a 2-hexadecimal digit.

Example:

```
while !lookAhead(":") traceText("0x" + readByte() + " ");
```

Output:

```
0x69 0x64 0x65 0x6E 0x74 0x69 0x66 0x69 0x65 0x72
```

See also:

`peekChar` 4.5.9, `getLastReadChars` 4.5.6, `readBytes` 4.5.12, `readChar` 4.5.14, `readChars` 4.5.16, `readCharAsInt` 4.5.15, `readIdentifier` 4.5.17, `readLine` 4.5.21, `readAdaString` 4.5.10, `readNumber` 4.5.23, `readPythonString` 4.5.24, `readString` 4.5.25, `readWord` 4.5.27

4.5.13 readBytes

- function **readBytes(**`length` : *int***)** : *string*

Parameter	Type	Description
length	*int*	number of bytes to read

Returns the sequence of `length` bytes read at the current file position, or an empty string if the end of file has been reached. If succeeded, the position of the input file moves just after.

The sequence of bytes is returned as a concatenation of 2-hexadecimal digits.

Example:

```
traceLine("6 first bytes = 0x" + readBytes(6));
```

Output:

```
6 first bytes = 0x6964656E7469
```

See also:

`peekChar` 4.5.9, `getLastReadChars` 4.5.6, `readByte` 4.5.11, `readChar` 4.5.14, `readChars` 4.5.16, `readCharAsInt` 4.5.15, `readIdentifier` 4.5.17, `readLine` 4.5.21, `readAdaString` 4.5.10, `readNumber` 4.5.23, `readPythonString` 4.5.24, `readString` 4.5.25, `readWord` 4.5.27

4.5.14 readCChar

- function **readCChar()** : *string*

  Returns the **C-like** constant character read at the current file position. A C-like character stands between single quotes and admits the escape character **r**. If succeeded, the position of the input file moves to the trailing single quote.

4.5.15 readChar

- function **readChar()** : *string*

  Returns the character read at the current file position, or an empty string if the end of file has been reached. If succeeded, the position of the input file moves to the next character.

  Example:

  ```
  while !lookAhead(":") traceText(readChar());
  ```

  Output:

  ```
  identifier
  ```

  See also:

  ```
  peekChar 4.5.9, getLastReadChars 4.5.6, readByte 4.5.11, readBytes 4.5.12,
  readChars 4.5.16, readCharAsInt 4.5.15, readIdentifier 4.5.17, readLine 4.5.21,
  readAdaString 4.5.10, readNumber 4.5.23, readPythonString 4.5.24, readString
  4.5.25, readWord 4.5.27
  ```

4.5.16 readCharAsInt

- function **readCharAsInt()** : *int*

  Returns the ASCII value of character read at the current file position, or a negative number −1 if the end of file has been reached. If succeeded, the position of the input file moves to the next character.

  Example:

  ```
  traceLine("we move to the end of a line,");
  readNextText("$");
  traceLine("so carriage return or newline is " +
  readCharAsInt());
  ```

  Output:

  ```
  we move to the end of a line,
  so carriage return or newline is 13
  ```

  See also:

  ```
  peekChar 4.5.9, getLastReadChars 4.5.6, readByte 4.5.11, readBytes 4.5.12,
  readChar 4.5.14, readChars 4.5.16, readIdentifier 4.5.17, readLine 4.5.21,
  readAdaString 4.5.10, readNumber 4.5.23, readPythonString 4.5.24, readString
  4.5.25, readWord 4.5.27
  ```

4.5.17 readChars

- function **readChars(**length **:** *int***) :** *string*

Parameter	Type	Description
length	*int*	number of characters to read

Returns the sequence of length characters read at the current file position, or an empty string if the end of file has been reached. If succeeded, the position of the input file moves just after.

Example:

```
traceLine("6 first characters = '" + readChars(6) + "'");
```

Output:

```
6 first characters = 'identi'
```

See also:

peekChar 4.5.9, getLastReadChars 4.5.6, readByte 4.5.11, readBytes 4.5.12, readChar 4.5.14, readCharAsInt 4.5.15, readIdentifier 4.5.17, readLine 4.5.21, readAdaString 4.5.10, readNumber 4.5.23, readPythonString 4.5.24, readString 4.5.25, readWord 4.5.27

4.5.18 readIdentifier

- function **readIdentifier() :** *string*

Returns the *identifier* token read at the position of the input file, or an empty string if it doesn't match.

An *identifier* begins with an alphabetical character (letter without accent) or an underscore and may be followed by any of them or by digits.

Example:

```
traceLine("we jump just before the identifier:");
readNextText("identifier:  ");
traceLine("identifier = '" + readIdentifier() + "'");
```

Output:

```
we jump just before the identifier:
identifier = '_potatoes41'
```

See also:

peekChar 4.5.9, getLastReadChars 4.5.6, readByte 4.5.11, readBytes 4.5.12, readChar 4.5.14, readChars 4.5.16, readCharAsInt 4.5.15, readLine 4.5.21, readAdaString 4.5.10, readNumber 4.5.23, readPythonString 4.5.24, readString 4.5.25, readWord 4.5.27

4.5.19 readIfEqualTo

- function **readIfEqualTo(**text **:** *string***) :** *bool*

Parameter	Type	Description
text	*string*	a sequence of characters to match

Checks whether the next characters of the input stream match with the string passed to argument `text`. If so, the function returns `true` and the position of the input stream moves just after.

Example:

```
// we move further into the input file,
// just after 'C++:  '
readNextText("C++:   ");
local iPosition = getInputLocation();
traceLine("readIfEqualTo('/*') = '" + readIfEqualTo("/*") +
"'");
if iPosition == getInputLocation() error("The file position
should have moved after '/*'!");
```

Output:

```
readIfEqualTo('/*') = 'true'
```

See also:

`lookAhead` 4.5.8, `readIfEqualToIgnoreCase` 4.5.20, `readIfEqualToIdentifier` 4.5.19

4.5.20   readIfEqualToIdentifier

- function **readIfEqualToIdentifier**(`identifier` : *string*) : *bool*

Parameter	Type	Description
`identifier`	*string*	an *identifier* is a string composed of letters and underscores ; digits are admitted too, except at the first place

Checks whether the next characters of the input stream match with the *identifier* passed to argument. If so, the function returns `true` and the position of the input stream moves just after.

This function warrants that the character just before the beginning of the *identifier* into the input stream is neither a letter nor a digit nor an underscore, to assure that the identifier really starts at the current position of the input stream.

Example:

```
traceLine("readIfEqualTo('ident') = '" + readIfEqualTo("ident")
+ "'");
traceLine("readIfEqualTo('identifier') = '" +
readIfEqualTo("identifier") + "'");
```

Output:

```
readIfEqualTo('ident') = 'true'
readIfEqualTo('identifier') = ''
```

See also:

`lookAhead` 4.5.8, `readIfEqualTo` 4.5.18, `readIfEqualToIgnoreCase` 4.5.20

4.5.21 readIfEqualToIgnoreCase

- function **readIfEqualToIgnoreCase**(text : *string*) : *bool*

Parameter	Type	Description
text	*string*	a sequence of characters to match

Checks whether the next characters of the input stream match with the string passed to argument text, ignoring the case. If so, the function returns true and the position of the input stream moves just after.

Example:

```
traceLine("readIfEqualToIgnoreCase('IDENTIFIER') = '" +
readIfEqualToIgnoreCase("IDENTIFIER") + "'");
if !readIfEqualTo(":") error("':'  expected after matching with
'IDENTIFIER'!");
```

Output:

```
readIfEqualToIgnoreCase('IDENTIFIER') = 'true'
```

See also:

lookAhead 4.5.8, readIfEqualTo 4.5.18, readIfEqualToIdentifier 4.5.19

4.5.22 readLine

- function **readLine**(text : *stringref*) : *bool*

Parameter	Type	Description
text	*stringref*	a variable that will contain the line

Reads the next line, starting at the current position of the input file, and puts it into parameter text. Characters '**\r**' or '**\n**' are ignored, and the position points to the beginning of the next line or at the end of file if reached.

If succeeded, the function returns true.

Example:

```
traceLine("Reads the 2 first lines:");
local sLine;
if !readLine(sLine) error("line 1 expected, instead of end of
file");
traceLine("\t" + sLine);
if !readLine(sLine) error("line 2 expected, instead of end of
file");
traceLine("\t" + sLine);
```

Output:

```
Reads the 2 first lines:
    identifier:  _potatoes41$
    numbers:  42 23.45e6
```

See also:

4.5.23   readNextText

- function **readNextText(**`text` **:** *string***) :** *bool*

Parameter	Type	Description
`text`	*string*	a sequence of characters to find

Looks for the next occurrence of the expression given by argument `text`, starting at the current position of the input file.

If succeeded, the position moves just after the expression given by `text` and the function returns `true`.

Example:

```
traceLine("position of the input stream = " +
getInputLocation());
if !readNextText("word:") error("where is 'word:'?");
traceLine("we jump to 'word:', and the new position is " +
getInputLocation());
```

Output:

```
position of the input stream = 0
we jump to 'word:', and the new position is 119
```

See also:

`readUptoJustOneChar` 4.5.26

4.5.24   readNumber

- function **readNumber(**`number` **:** *doubleref***) :** *bool*

Parameter	Type	Description
`number`	*doubleref*	a variable that will contain the number read into the input stream

Reads a number at the current position of the input stream, and puts it into the variable `number`. A number is either an integer or a floating-point representation as encountered ordinary.

If succeded, the position moves just after the token and the function returns `true`.

Example:

```
traceLine("we jump just before the numbers:");
readNextText("numbers:  ");
local dNumber;
if !readNumber(dNumber) error("integer expected!");
traceLine("integer = " + dNumber);
```

```
skipBlanks();
if !readNumber(dNumber) error("double expected!");
traceLine("double = " + dNumber);
```

Output:

```
we jump just before the numbers:
integer = 42
double = 23450000
```

See also:

peekChar 4.5.9, getLastReadChars 4.5.6, readByte 4.5.11, readBytes 4.5.12,
readChar 4.5.14, readChars 4.5.16, readCharAsInt 4.5.15, readIdentifier 4.5.17,
readLine 4.5.21, readAdaString 4.5.10, readPythonString 4.5.24, readString
4.5.25, readWord 4.5.27

4.5.25 readPythonString

- function **readPythonString**(text : *stringref*) : *bool*

Parameter	Type	Description
text	*stringref*	a variable that will contain the string literal extracted from the input stream

Reads a string literal as defined in *Python*, a scripting language. Notably, it accepts triple-quoted
strings.

If succeeded, the position moves just after the trailing double quote and the function returns true.

See also:

peekChar 4.5.9, getLastReadChars 4.5.6, readByte 4.5.11, readBytes 4.5.12,
readChar 4.5.14, readChars 4.5.16, readCharAsInt 4.5.15, readIdentifier 4.5.17,
readLine 4.5.21, readAdaString 4.5.10, readNumber 4.5.23, readString 4.5.25,
readWord 4.5.27

4.5.26 readString

- function **readString**(text : *stringref*) : *bool*

Parameter	Type	Description
text	*stringref*	a variable that will contain the string literal extracted from the input stream

Reads a string literal surrounded by double quotes, and where escape sequences are written as
presented in function composeCLikeString() (4.3.25). The token is then put into the vari-
able passed to argument text, without double quotes and after converting the escape sequences
into their ASCII representation.

If succeeded, the position moves just after the trailing double quote and the function returns true.

Example:

```
traceLine("we jump just before the string:");
readNextText("string:  ");
```

```
local sText;
if !readString(sText) error("constant string expected!");
traceLine("string = '" + sText + "'");
```

Output:

```
we jump just before the string:
string = 'a C-like string that accepts backslash-escape
sequences'
```

See also:

peekChar 4.5.9, getLastReadChars 4.5.6, readByte 4.5.11, readBytes 4.5.12, readChar 4.5.14, readChars 4.5.16, readCharAsInt 4.5.15, readIdentifier 4.5.17, readLine 4.5.21, readAdaString 4.5.10, readNumber 4.5.23, readPythonString 4.5.24, readWord 4.5.27

4.5.27   readUptoJustOneChar

- function **readUptoJustOneChar**(oneAmongChars **:** *string***) :** *string*

Parameter	Type	Description
oneAmongChars	*string*	a set of characters

Reads the input stream up to encountering a character that belongs to the parameter oneAmongChars, and returns the sequence of characters read.

The position of the input stream points to the first character that belongs to the parameter oneAmongChars. Calling readChar() or readIfEqualTo() just after allows determining what character, amongst those put into oneAmongChars, was encountered first.

Example:

```
traceLine("readUptoJustOneChar('$_:')  = '" +
readUptoJustOneChar("$_:") + "'");
```

Output:

```
readUptoJustOneChar('$_:')  = 'identifier'
```

See also:

readNextText 4.5.22

4.5.28   readWord

- function **readWord**() **:** *string*

Returns the *word* token read at the position of the input file, or an empty string if it doesn't match.

CODEWORKER understands a *word* token as a sequence of alphabetical characters, including letters with an accent as existing in French or Germany, or underscores or digits.

Example:

```
traceLine("we jump just before the word:");
readNextText("word:  ");
traceLine("readWord() = '" + readWord() + "'");
```

Output:

```
we jump just before the word:
readWord() = '1'
```

See also:

`peekChar` 4.5.9, `getLastReadChars` 4.5.6, `readByte` 4.5.11, `readBytes` 4.5.12, `readChar` 4.5.14, `readChars` 4.5.16, `readCharAsInt` 4.5.15, `readIdentifier` 4.5.17, `readLine` 4.5.21, `readAdaString` 4.5.10, `readNumber` 4.5.23, `readPythonString` 4.5.24, `readString` 4.5.25

4.5.29   setInputLocation

- procedure **setInputLocation**(`location` : *int*)

Parameter	Type	Description
`location`	*int*	points to a position of the input stream

This procedure moves the position of the input stream elsewhere. The position starts at 0.

Example:

```
traceLine("we jump to identifier 'potatoes' at position 12");
setInputLocation(12);
if !readIfEqualToIdentifier("_potatoes41") error("identifier
'_potatoes41' expected");
```

Output:

```
we jump to identifier 'potatoes' at position 12
```

Deprecated form: `setLocation` has disappeared since version *3.7.1*

See also:

`goBack` 4.5.7, `getInputLocation` 4.5.5

4.5.30   skipBlanks

- function **skipBlanks()** : *bool*

Ignores all blank characters (ASCII codes between 0x00 and 0x20 both included, range that comprises newline, carriage-return, tabulations and spaces). The current file position moves to the first character that isn't a blank.

Example:

```
readNextText("blanks:   ");
traceLine("we skip blank characters");
skipBlanks();
traceText("now, we read the string token:  ");
local sText;
if !readString(sText) error("constant string expected");
traceLine("'" + sText + "'");
```

Output:

we skip blank characters
now, we read the string token: 'blanks are ignored'

See also:

`skipSpaces` 4.5.34, `skipEmptyCpp` 4.5.30, `skipEmptyCppExceptDoxygen` 4.5.31, `skipEmptyHTML` 4.5.32, `skipEmptyLaTeX` 4.5.33

4.5.31   skipEmptyCpp

- function **skipEmptyCpp()** : *bool*

  Ignores all blank characters and all C++/JAVA-like comments. The current file position moves to the first character that must be kept.

  Example:

```
readNextText("C++:  ");
traceLine("we skip C++ empty tokens");
skipEmptyCpp();
traceText("now, we read the string token:  ");
local sText;
if !readString(sText) error("constant string expected");
traceLine("'" + sText + "'");
```

  Output:

```
we skip C++ empty tokens
now, we read the string token:  'blanks and C++ comments are
ignored'
```

  See also:

  `skipBlanks` 4.5.29, `skipSpaces` 4.5.34, `skipEmptyCppExceptDoxygen` 4.5.31, `skipEmptyHTML` 4.5.32, `skipEmptyLaTeX` 4.5.33

4.5.32   skipEmptyCppExceptDoxygen

- function **skipEmptyCppExceptDoxygen()** : *bool*

  Ignores all blank characters and all C++/JAVA-like comments, except when the comment conforms to the *Doxygen* format. The current file position moves to the first character that must be kept.

  Example:

```
readNextText("C++:  ");
traceLine("we skip C++ empty tokens, except Doxygen comments");
skipEmptyCppExceptDoxygen();
traceText("now, we read the string token:  ");
local sText;
if !readString(sText) error("constant string expected");
traceLine("'" + sText + "'");
```

  Output:

```
we skip C++ empty tokens, except Doxygen comments
now, we read the string token:  'blanks and C++ comments are
ignored'
```

See also:

```
skipBlanks 4.5.29, skipSpaces 4.5.34, skipEmptyCpp 4.5.30, skipEmptyHTML
4.5.32, skipEmptyLaTeX 4.5.33
```

4.5.33 skipEmptyHTML

- function **skipEmptyHTML() :** *bool*

  Ignores all blank characters and all HTML/XML-like comments. The current file position moves to the first character that must be kept.

  Example:

```
readNextText("HTML: ");
traceLine("we skip HTML empty tokens");
skipEmptyHTML();
traceText("now, we read the string token:  ");
local sText;
if !readString(sText) error("constant string expected");
traceLine("'" + sText + "'");
```

  Output:

```
we skip HTML empty tokens
now, we read the string token:  'blanks and HTML comments are
ignored'
```

  See also:

```
skipBlanks      4.5.29,      skipSpaces      4.5.34,      skipEmptyCpp      4.5.30,
skipEmptyCppExceptDoxygen 4.5.31, skipEmptyLaTeX 4.5.33
```

4.5.34 skipEmptyLaTeX

- function **skipEmptyLaTeX() :** *bool*

  Ignores all LaTeX comments. A LaTeX comment is announced by the character `percent` and finishes at the end of the line. The current file position moves to the first character that must be kept.

  Example:

```
readNextText("LaTeX: ");
traceLine("we skip LaTeX comments");
skipEmptyLaTeX();
traceText("now, we read the string token:  ");
local sText;
if !readString(sText) error("constant string expected");
traceLine("'" + sText + "'");
```

  Output:

```
we skip LaTeX comments
now, we read the string token:  'blanks must be skipped
explicitly'
```

  See also:

skipBlanks `4.5.29`, skipSpaces `4.5.34`, skipEmptyCpp `4.5.30`,
skipEmptyCppExceptDoxygen `4.5.31`, skipEmptyHTML `4.5.32`

4.5.35 skipSpaces

- function **skipSpaces()** : *bool*

  Ignores all space and tabulation characters. The current file position moves to the first character that isn't a space or a tabulation.

  Example:

```
readNextText("spaces:");
traceLine("we skip spaces and tabulations");
skipSpaces();
traceText("now, we read the string token:  ");
local sText;
if !readString(sText) error("constant string expected");
traceLine("'" + sText + "'");
```

  Output:

```
we skip spaces and tabulations
now, we read the string token:  'spaces are ignored'
```

  See also:

  skipBlanks `4.5.29`, skipEmptyCpp `4.5.30`, skipEmptyCppExceptDoxygen `4.5.31`,
  skipEmptyHTML `4.5.32`, skipEmptyLaTeX `4.5.33`

4.6 Syntax and instructions for generating source code

A script that must be processed for source code generation is called a *pattern script* in the CODE-WORKER vocabulary. It exists three ways to generate a file:

- the classical **generation mode** is used to let the script produce the most part of the output. The output file is rewritten completely. Only some areas called *protected areas* in the vocabulary of CODEWORKER are preserved in the file. This philosophy has been adopted by some modeling tools that generate a skinny skeleton copiously interspersed with areas intended to the developer.

- **expansion mode** is used when the file is mainly written by hand, but small portions need to be generated. The points where to insert the code are known as *markups* in the vocabulary of CODEWORKER . Visual C++ changes source code like it via the *Class Wizard*.

- **translation mode** is used when both parsing and source code generation are required to process a file. It arrives when a file must be rewritten in a different syntax, or when the positions the code must be inserted depend on a strategy that is determined by the parsing. For example, a script should add a trace at the beginning of each function body of a JAVA or C++ source code. To do that, parsing will allow discovering function bodies, and source code generation will write the C++ or JAVA code that implements the trace.

A *pattern script*, except in *translation mode*, begins with a sequence of characters exactly like they must be written into the output file, up to it encounters special character **'@'** or JSP-like tag **'<%'**. Then it

---

swaps into script mode, and everything is interpreted as script instructions, up to the special character `'@'` or the JSP-like tag `'%>'` are encountered. The content of the script file is again understood as a sequence of characters to write into the output file, up to the next special character. And it continues swapping from a mode to another...

For convenience, the script mode might just be restrained to an expression (often a variable expression) whose value is written into the output file.

Expanding a file consists of generating code to some determined points of the file. These points are called `markups` and are noted **##markup##**"*name-of-the-markup*", surrounded by comment delimiters.

For example, a valid markup inlayed in a C++ file could be:
`//##markup##"factory"`
and a valid markup inlayed in an HTML file could be:
`<!- -##markup##"classes"- ->`

A *pattern* script intended to expand code is launched thanks to the procedure `expand` that expects three parameters:

- the first one is the file name of the script,

- the second one is the current context of execution that will be accessed via the `this` keyword into the script,

- the last one is the name of the file to expand,

Each time CODEWORKER will encounter a markup, it will call the *pattern* script that will decide how to populate it. The code generated by the *pattern* script for this markup is surrounded by tags **##begin##**"*name-of-the-markup*" and **##end##**"*name-of-the-markup*", automatically added by the interpreter. If some protected areas were put into the generated code, they are preserved the next time the expansion is required.

Note that CODEWORKER doesn't change what is written outside the markups and their begin/end delimiters.

A *script* that is intended to work on *translation mode* expects first a BNF-like description as presented at section 4.3.216. As for any kind of *BNF-driven* script, *procedural-driven* script may be inlayed in braces after symbol `'=>'`. This compound statement may contain a subset of *pattern script*, as described in the precedent paragraph, which will take in charge of generating code into the output file. Note that the flow of execution enters into the compound statement in *script mode*.

Such as for parsing, it exists some functions to handle a position into the output stream. However, the principle is quite different, insofar as the current position of the output stream cannot be changed and always points to the end.

A position is called a *floating location* and has an ID. A *floating location* is used for overwriting or for inserting text to a point of the stream that has already been generated. While generating a C++ body for example, it may be interesting to insert the `'#include'` preprocessor directive as references to other headers are discovered during the iteration of the parse tree.

The procedure `newFloatingLocation` allows attaching a position to an ID, which represents the name of the location. The function `getFloatingLocation` returns the position attached to a given *floating location* ID.

Inserting text at a position leads to shift all *floating locations* that are pointing to, or after, the insertion point. The offset corresponds to the size of the text. So, it is called a *floating location* because the position assigned initially to the ID might change in the future.

You'll find below a list of all built-in functions or procedures that may be used into a *pattern script*, as well as typical preprocessor directives.

4.6.1 Preprocessor directive: coverage recording

A functionality has been added to code generation, to know where the output comes from. In Code-Worker, an output file is generated by a template-based script. The directive `#coverage` asks for the recording of every script position giving rise to a piece of the output file.

This directive is located anywhere in the script to study, and requires a variable the code generation engine will populate with coverage data. The variable will be a list of sorted segments, entirely determined by their starting position in the output and by the position of the corresponding script *instruction*. These positions are respectively stored in attributes **output** and **script**.

An adding information is assigned to the node representing the segment. It specifies the type of script *instruction*, belonging to one of the following values:

- **"R"**: rough text,

- **"W"**: expression, variable or call to a `writeText()`-family procedure,

- **"I"**: call to a `insertText()`-family procedure,

- **"O"**: call to `overwritePortion()` procedure,

Example:
rough text @`this.name`**@ EOL**
@
`#coverage(project.coverage)`

Let say that *this.name* is worth "VARIABLE_CONTENT". The script generates the following output file:
```
rough text VARIABLE_CONTENT EOL
```

The variable `project.coverage` is then worth the following list:
```
["0"] = "R"
  |-+
    script = 12
    output = 0
["1"] = "W"
  |-+
    script = 21
    output = 11
["2"] = "R"
  |-+
    script = 29
    output = 27
```

4.6.2 Aspect-Oriented Programming and template-based scripts

This section will be extended later. First, we'll just focus on features turning around AOP in Code-Worker.

A template-based script can indicate some joint points during the generation process. A joint point represents a remarkable place in the output stream, like the declaration of attributes or the body of a

---

method. The developer is free to create as many joint point as needed. He gives to them the meaning that he wishes.

The syntax of a joint point looks like:
jointpoint [**iterate**]? *name* [**(** *context* **)**]? **;**
or
jointpoint [**iterate**]? *name* [**(** *context* **)**]? *instruction*

context is a variable expression. If *context* is not specified, it is worth to **this**.

When the interpreter encounters a joint point, it checks the existence of the variable *context*. If the variable exists, the interpreter looks for actions to execute before, around and after the joint point.

These actions are referred to as *advices*. They normally intend for generating text at the place of some particular joint points. To determine what are joint points on which an advice must apply, the developer has to define point cuts. A point cut takes the form of a boolean expression attached to the advice.

The syntax of an advice looks like: **advice advice-type :** *pointcut instruction*
with: **advice-type ::= before | around | after | before_iteration |**
around_iteration | after_iteration

Note that *pointcut* is a boolean expression whose variable scope contains two local variables:

- **jointpoint**: the name of the joint point currently processed,

- **context**: a reference to the *context* variable passed to the joint point,

When the interpreter considers a joint point, it first looks for each advice of type **before** where the point cut matches. Then it executes them in the order they were implemented. Next, it looks for each advice working around the joint point (of type **around**) and executes them. When the interpreter leaves a joint point, it executes each advice of type **after** where the point cut matches.

An advice can execute the body of a joint point at any time, eventually changing the current context, using the directive **#jointpoint**:
#jointpoint [**(** *context* **)**]?

If the option **iterate** is requested, the joint point works on an array. If the array is empty, the interpreter bypasses the joint point. If not, the interpreter iterates each element of the array, looking for advices before, around and after each iteration. The corresponding advice types are referred to as **before_iteration**, **around_iteration** and **after_iteration**.

Example of a jointpoint working on an array:

```
 // file "Documentation/AOP-example1.cwt":
Separation of concerns in CodeWorker:
@
// a list of method declarations (names only)
insert this.methods["display"].name = "display";
insert this.methods["delete"].name = "delete";
insert this.methods["visit"].name = "visit";

// a joint point announcing the implementation
// of methods
jointpoint iterate methods(this.methods) {
    // generates the skeleton of a method
    @ void @this.name@() {}
@
```

```
        }

        @...  This is the end of the test
        @

        // Implementation of an aspect to apply on methods:
        // several advices

        // first advice:  called while entering the joint point
        advice before :  jointpoint == "methods" {
           @// beginning of methods
        @
        }

        // advice to apply around each method iteration
        advice around_iteration :  jointpoint == "methods" {
           @ // BEGIN around iteration
        @
           #jointpoint
           @ // OUT around iteration
        @
        }

        // last advice:  called while leaving the joint point
        advice after :  jointpoint == "methods" {
           @// end of methods
        @
           traceObject(this, 10);
        }
```

It generates the following output:

```
         // file "Documentation/AOP-example1.txt":
        testing separation of concerns in CodeWorker:

        // beginning of methods
           // BEGIN around iteration
           void display() {}
           // OUT around iteration
           // BEGIN around iteration
           void delete() {}
           // OUT around iteration
           // BEGIN around iteration
           void visit() {}
           // OUT around iteration
        // end of methods

        This is the end of the test
```

4.6.3   allFloatingLocations

- procedure **allFloatingLocations(**list **:** *tree***)**

Parameter	Type	Description
list	*tree*	floating location names and their position

Populates the argument list with all floating location registered to the current output stream, such as the entry key is the floating location name and the entry value is the position in the stream.

See also:

getFloatingLocation 4.6.11, existFloatingLocation 4.6.9, getOutputLocation 4.6.16, newFloatingLocation 4.6.25, removeFloatingLocation 4.6.29, setFloatingLocation 4.6.32, setOutputLocation 4.6.33

4.6.4 attachOutputToSocket

- procedure **attachOutputToSocket(**socket **:** *int***)**

Parameter	Type	Description
socket	*int*	a client socket descriptor

Joins the output stream of a *template-based* or *translation* script so that to send the generated text to the socket.

The generated text is sent at the end of the complete script execution.

See also:

detachOutputFromSocket 4.6.7, flushOutputToSocket 4.6.10, createINETClientSocket 4.3.36, createINETServerSocket 4.3.37, acceptSocket 4.3, attachInputToSocket 4.5, detachInputFromSocket 4.5.3, receiveBinaryFromSocket 4.3.153, receiveFromSocket 4.3.154, receiveTextFromSocket 4.3.155, sendTextToSocket 4.3.180, sendBinaryToSocket 4.3.178, closeSocket 4.3.18

4.6.5 countOutputCols

- function **countOutputCols() :** *int*

Determines the column number in the line where the output stream cursor points to.

See also:

countOutputLines 4.6.5

4.6.6 countOutputLines

- function **countOutputLines() :** *int*

Determines the line number where the output stream cursor points to.

See also:

countOutputCols 4.6.4

4.6.7 decrementIndentLevel

- function **decrementIndentLevel(**`level` **:** *int***) :** *bool*

Parameter	Type	Description
`level`	*int*	**default value:** *1* depth of indentation to remove

Decrements the indentation level, used to indenting output files automatically while writing.

The function returns `false` if the *level* parameter is greater than the current indentation depth. If the function returns `false`, the indentation level is worth zero and it disables the indent-mode. Otherwise, each time a text will have to be written at the beginning of a line, the line will be indented, depending on the indentation level.

Call the function `incrementIndentLevel()` to increase the indentation.

See also:

`incrementIndentLevel` 4.6.19

4.6.8 detachOutputFromSocket

- procedure **detachOutputFromSocket(**`socket` **:** *int***)**

Parameter	Type	Description
`socket`	*int*	a client socket descriptor

Disconnects the output stream of a parsing script from a socket stream. You should have join the socket to the output stream before, via the procedure `attachOutputToSocket()`.

To call only if you have changed your mind and don't want the generated text to be sent to the socket at the end of the *template-based* script execution anymore.

See also:

`createINETClientSocket` 4.3.36, `createINETServerSocket` 4.3.37, `acceptSocket` 4.3, `attachInputToSocket` 4.5, `detachInputFromSocket` 4.5.3, `attachOutputToSocket` 4.6.3, `receiveBinaryFromSocket` 4.3.153, `receiveFromSocket` 4.3.154, `receiveTextFromSocket` 4.3.155, `sendTextToSocket` 4.3.180, `sendBinaryToSocket` 4.3.178, `closeSocket` 4.3.18, `flushOutputToSocket` 4.6.10

4.6.9 equalLastWrittenChars

- function **equalLastWrittenChars(**`text` **:** *string***) :** *bool*

Parameter	Type	Description
`text`	*string*	string value to compare

Compare the `text` argument with last characters written in the output file.

Example:

```
let's write some characters@
traceLine("equalLastWrittenChars(\"characters\") = '" +
equalLastWrittenChars("characters") + "'");
```

Output:

```
equalLastWrittenChars("characters") = 'true'
```

Generated text:

```
let's write some characters
```

See also:

`getLastWrittenChars` 4.6.12

4.6.10   existFloatingLocation

- function **existFloatingLocation**(key : *string*, parent : *bool*) : *bool*

Parameter	Type	Description
key	*string*	name of a floating position into the output stream
parent	*bool*	while expanding a file, search into precedent markup area if this floating location exists or not

Returns `true` if the *floating location*, whose name is passed to argument key, already exists.

If required (parameter parent set to `true`) and if the current output file is being expanded, the function searches the floating location in precedent visited markup areas.

Example:

```
Roger Rabbit@
newFloatingLocation("Roger Rabbit");
@ doesn't like spinash@
traceLine("Does the floating location 'Roger Rabbit' exists?  '"
+ existFloatingLocation("Roger Rabbit", false) + "'");
```

Output:

```
Does the floating location 'Roger Rabbit' exists?  'true'
```

Generated text:

```
Roger Rabbit doesn't like spinash
```

See also:

```
getFloatingLocation     4.6.11,      allFloatingLocations     4.6.2,
getOutputLocation       4.6.16,      newFloatingLocation      4.6.25,
removeFloatingLocation  4.6.29,      setFloatingLocation      4.6.32,
setOutputLocation 4.6.33
```

4.6.11   flushOutputToSocket

- function **flushOutputToSocket**(socket : *int*) : *bool*

Parameter	Type	Description
socket	*int*	a client socket descriptor

Sends to a socket the complete text or binary data already generated by the *template-based* or *translation* script. The function then purges the output stream.

This function has no link with `attachOutputToSocket()`. It only requires that the socket descriptor exists and is opened correctly.

See also:

`attachOutputToSocket` 4.6.3, `detachOutputFromSocket` 4.6.7, `createINETClientSocket` 4.3.36, `createINETServerSocket` 4.3.37, `acceptSocket` 4.3, `attachInputToSocket` 4.5, `detachInputFromSocket` 4.5.3, `receiveBinaryFromSocket` 4.3.153, `receiveFromSocket` 4.3.154, `receiveTextFromSocket` 4.3.155, `sendTextToSocket` 4.3.180, `sendBinaryToSocket` 4.3.178, `closeSocket` 4.3.18

4.6.12 getFloatingLocation

- function **getFloatingLocation**(`key` : *string*) : *int*

Parameter	Type	Description
`key`	*string*	name of an existing floating position into the output stream

Returns the position into the output stream attached to a *floating location* whose name is passed to argument `key`.

The function raises an error if the floating location doesn't exist or is present in a precedent markup area of a being-expanded file, but not in the current markup area.

Example:

```
Roger Rabbit@
newFloatingLocation("Roger Rabbit");
@ doesn't like spinash@
traceLine("Position just after 'Roger Rabbit' is " +
getFloatingLocation("Roger Rabbit"));
```

Output:

```
Position just after 'Roger Rabbit' is 12
```

Generated text:

```
Roger Rabbit doesn't like spinash
```

See also:

`allFloatingLocations` 4.6.2, `existFloatingLocation` 4.6.9, `getOutputLocation` 4.6.16, `newFloatingLocation` 4.6.25, `removeFloatingLocation` 4.6.29, `setFloatingLocation` 4.6.32, `setOutputLocation` 4.6.33

4.6.13 getLastWrittenChars

- function **getLastWrittenChars**(`NbChars` : *int*) : *string*

Parameter	Type	Description
`NbChars`	*int*	how many characters to ask for

Returns the last characters written in the output file, up to `NbChars` or less if the beginning of the file is encountered too early.

Example:

```
let's write some characters@
traceLine("getLastWrittenChars(10) = '" + getLastWrittenChars(10)
+ "'");
```

Output:

```
getLastWrittenChars(10) = 'characters'
```

Generated text:

```
let's write some characters
```

See also:

```
equalLastWrittenChars 4.6.8
```

4.6.14   getMarkupKey

- function **getMarkupKey() :** *string*

This function must be called into a pattern script that runs in *expansion mode*. It returns the current markup key where code has to be expanded now.

Example:

```
@
if getMarkupKey() == "examples" {
    @A little example?@
} else {
    @We are doing the LaTeX documentation in expansion mode!@
    traceLine("current markup of the LaTeX documentation = '" +
getMarkupKey() + "'");
}
```

Output:

```
current markup of the LaTeX documentation = 'generation script
functions'
```

Generated text:

```
We are doing the LaTeX documentation in expansion mode!
```

Deprecated form: `getMarkerKey` has disappeared since version *2.14*

See also:

```
getMarkupValue 4.6.14
```

4.6.15   getMarkupValue

- function **getMarkupValue() :** *string*

This function must be called into a pattern script that runs in *expansion mode*. It returns the current markup value, if any, put between tags '##data##' after declaring the markup key.

Example - hand-typed code to expand:

```
...    extension of C++/Java:  implements a 'switch' on strings
//##markup##"switch(sText)"
//##data##
//Product
//RentalItem
//Customer
//RentalOrder
//Figurine
//Movie
//DVD
//VideoStore
//##data##
...  next code
```

Example - the script:

```
{
    int iHashCode = 0;
    std::string sKey = @coreString(getMarkupKey(), 7, 1)@;
    for (int i = 0; i < sKey.length(); i++) {
        unsigned char c = sKey[i];
        iHashCode = (31*iHashCode + (c%31)) % 64000000;
    }
    switch(iHashCode) {
@
local sData = getMarkupValue();
while sData {
    local iIndex = sData.findString('\n');
    if $iIndex < 0$ || !sData.startString("//") error("syntax
error");
    local sKey = sData.midString(2, $iIndex - 2$);
    if sKey.endString('\r') set sKey = sKey.rsubString(1);
    local iHashCode = 0;
    local i = 0;
    while $i < sKey.length()$ {
        local c = sKey.charAt(i);
        iHashCode = $(31*iHashCode + (c.charToInt()%31)) %
64000000$;
        increment(i);
    }
    @ case @iHashCode@:  // "@sKey@"
@
    setProtectedArea("case \"" + sKey + "\":");
    set sData = sData.subString($iIndex + 1$);
}
@ default:
@
setProtectedArea("default:");
@ }
}
@
```

Generated text:

```
...  extension of C++/Java:  implements a 'switch' on strings
//##markup##"switch(sText)"
//##data##
//Product
//RentalItem
//Customer
//RentalOrder
//Figurine
//Movie
//DVD
//VideoStore
//##data##
//##begin##"switch(sText)"
{
                int iHashCode = 0;
                std::string sKey = sText;
                for (int i = 0; i < sKey.length(); i++) {
                    unsigned char c = sKey[i];
                    iHashCode = (31*iHashCode + (c%31)) % 64000000;
                }
                switch(iHashCode) {
                    case 17133617:  // "Product"
            //##protect##"case \"Product\":"
//##protect##"case \"Product\":"
        case 19256985:  // "RentalItem"
            //##protect##"case \"RentalItem\":"
//##protect##"case \"RentalItem\":"
        case 26793087:  // "Customer"
            //##protect##"case \"Customer\":"
//##protect##"case \"Customer\":"
        case 26446891:  // "RentalOrder"
            //##protect##"case \"RentalOrder\":"
//##protect##"case \"RentalOrder\":"
        case 20050752:  // "Figurine"
            //##protect##"case \"Figurine\":"
//##protect##"case \"Figurine\":"
        case 14413458:  // "Movie"
            //##protect##"case \"Movie\":"
//##protect##"case \"Movie\":"
        case 6516:  // "DVD"
            //##protect##"case \"DVD\":"
//##protect##"case \"DVD\":"
        case 56055430:  // "VideoStore"
            //##protect##"case \"VideoStore\":"
//##protect##"case \"VideoStore\":"
        default:
            //##protect##"default:"
//##protect##"default:"
    }
            }
            //##end##"switch(sText)"
```

```
...    next code
```

See also:

`getMarkupKey` 4.6.13

4.6.16 getOutputFilename

- function **getOutputFilename() :** *string*

Returns the path of the output file being generated.

4.6.17 getOutputLocation

- function **getOutputLocation() :** *int*

Returns the current file position for writing to the output stream.

Example:

```
I write 22 characters.@
traceLine("Current position to the output stream = " +
getOutputLocation());
```

Output:

```
Current position to the output stream = 22
```

Generated text:

```
I write 22 characters.
```

See also:

```
getFloatingLocation      4.6.11,      allFloatingLocations       4.6.2,
existFloatingLocation       4.6.9,      newFloatingLocation       4.6.25,
removeFloatingLocation      4.6.29,      setFloatingLocation       4.6.32,
setOutputLocation 4.6.33
```

4.6.18 getProtectedArea

- function **getProtectedArea(**`protection` **:** *string***) :** *string*

Parameter	Type	Description
`protection`	*string*	unique ID of a protected area

Returns the content of the protected area whose name is given by parameter `protection`. If the
protected area wasn't found into the ancient version of the output stream, and if it hasn't been put
yet via `setProtectedArea` into the output stream, it returns an empty string.

Example:

```
@
setProtectedArea("keep this code for me, please!");
if !getProtectedArea("keep this code for me, please!") {
    traceLine("you have never populated the protected area!");
}
```

4.6. Syntax and instructions for generating source code   279

Output:

```
you have never populated the protected area!
```

Generated text:

```
//##protect##"keep this code for me, please!"
//##protect##"keep this code for me, please!"
```

See also:

```
populateProtectedArea      4.6.27,      getProtectedAreaKeys      4.6.18,
remainingProtectedAreas      4.6.28,      removeProtectedArea      4.6.30,
setProtectedArea 4.6.34
```

4.6.19   getProtectedAreaKeys

- function **getProtectedAreaKeys(**`keys` **:** *tree***) :** *int*

Parameter	Type	Description
`keys`	*tree*	it will contain keys of all protected areas that were found into the output stream before the generation

Looks for all protected areas that were found into the output stream before processing the source code generation and puts their keys into the well-named argument `keys`.

Note: `keys` stores the ID of protected areas in the lexicographical order.

The function returns how many protected areas were found into the output stream. Both key and value of items are worth the protected area key. If the list wasn't declared as *local* or *global* or as belonging to `this`, a negative value is returned.

Example - precedent output generated:

```
My first protected area:
//##protect##"hand-typed code"
I don't want to loose this portion of text
after another generation!
//##protect##"hand-typed code"
My second protected area:
//##protect##"don't forget me, please!"
This portion of text is also very important.
//##protect##"don't forget me, please!"
Now, you can erase text, I don't care.
```

Example - the script:

```
My first protected area:
@
local listOfKeys;
local iNbKeys = getProtectedAreaKeys(listOfKeys);
traceLine("Number of preserved area at the beginning = " +
iNbKeys + ":");
foreach i in listOfKeys traceLine(" '" + i + "'");
setProtectedArea("hand-typed code");
traceLine("after removing and setting protected areas,");
traceLine("the function behaves the same");
```

```
removeProtectedArea("don't forget me, please!");
if $getProtectedAreaKeys(listOfKeys) != 2$ error("bad
behavior");
@Now, you can erase text, I don't care.
@
```

Output:

```
Number of preserved area at the beginning = 2:
    'don't forget me, please!'
    'hand-typed code'
after removing and setting protected areas,
the function behaves the same
```

Generated text:

```
My first protected area:
        //##protect##"hand-typed code"
I don't want to loose this portion of text
after another generation!
//##protect##"hand-typed code"
Now, you can erase text, I don't care.
```

See also:

4.6.20   incrementIndentLevel

- procedure **incrementIndentLevel(**level **:** *int***)**

Parameter	Type	Description
level	*int*	**default value:** *1* depth of indentation to add

Increments the indentation level, used to indenting output files automatically while writing.

If first call, it enables the indent-mode: each time a text will have to be written at the beginning of a line, the line will be indented, depending on the indentation level.

Call the function decrementIndentLevel() to decrease the indentation.

See also:

4.6.21   indentText

- function **indentText(**mode **:** *string***) :** *bool*

Parameter	Type	Description
mode	*string*	type of text to indent

Indents the output stream, conforming to a given type of file, to choose amongst:

- **C++**
- **JAVA**

More format will be recognized in the future.

The function returns `true` if the output stream needed to be indented, meaning that it has changed after processing the indentation.

Be careful that if you are expanding a file, only the expanded area will be indented. In that case, it is better to use `indentFile` once the generation has completed.

Example:

```
int f(int i) {
switch (i) {
case 1:
g(i + 1);
break;
case 2:
case 3:
if (i == 2) {
h();
}
g(i - 1);
break;
        }
}@
if indentText("C++") traceLine("the output stream wasn't
indented correctly");
```

Output:

```
the output stream wasn't indented correctly
```

Generated text:

```
int f(int i) {
    switch (i) {
        case 1:
            g(i + 1);
            break;
        case 2:
        case 3:
            if (i == 2) {
                h();
            }
            g(i - 1);
            break;
    }
}
```

See also:

`indentFile` 4.3.108

4.6.22 insertText

- procedure **insertText**(`location` : *int*, `text` : *string*)

Parameter	Type	Description
`location`	*int*	points to a position of the output stream
`text`	*string*	sequence of characters to insert

Inserts a sequence of characters passed to argument `text`, at the position of the output stream given by argument `location`. The position starts counting at 0. All *floating locations* that point to `location`, or after, are impacted by the insertion, and shift for an offset that is worth the size of the text to insert.

If the position isn't valid, negative or bigger than the end of the output stream, an error is raised.

In *expansion mode*, the position 0 points to the first character written for expansion and cannot exceed the last character written for expansion.

Generally, the position is given by the function `getFloatingLocation()`.

Example:

```
I'll drink a bottle of 'Margaux' year 1994@
newFloatingLocation("You'll be drunk!");
@ before smoking a cigar.@
traceLine("My glass is empty, let's try another bottle, year
1996");
insertText(getFloatingLocation("You'll be drunk!"), " and year
1996");
traceLine("My glass is empty once again, let's try another
bottle, year 2000");
insertText(getFloatingLocation("You'll be drunk!"), " and year
2000");
```

Output:

```
My glass is empty, let's try another bottle, year 1996
My glass is empty once again, let's try another bottle, year
2000
```

Generated text:

```
I'll drink a bottle of 'Margaux' year 1994 and year 1996 and
year 2000 before smoking a cigar.
```

See also:

`insertTextOnce` 4.6.22, `insertTextOnceToFloatingLocation` 4.6.23, `insertTextToFloatingLocation` 4.6.24, `overwritePortion` 4.6.26, `writeBytes` 4.6.35, `writeText` 4.6.36, `writeTextOnce` 4.6.37

4.6.23 insertTextOnce

- procedure **insertTextOnce**(`location` : *int*, `text` : *string*)

Parameter	Type	Description
`location`	*int*	points to a position of the output stream
`text`	*string*	sequence of characters to insert

Inserts a sequence of characters passed to argument `text`, at the position of the output stream given by argument `location`, only if the text has never been encountered previously by a `insertTextOnce()` or a `writeTextOnce()` procedure. The position starts counting at 0. All *floating locations* that point to `location`, or after, are impacted by the insertion, and shift for an offset that is worth the size of the text to insert.

If the position isn't valid, negative or bigger than the end of the output stream, an error is raised.

In *expansion mode*, the position 0 points to the first character written for expansion and cannot exceed the last character written for expansion.

Generally, the position is given by the function `getFloatingLocation()`.

Example:

```
@
newFloatingLocation("include files");
@
void f(const std::string& s) {
...
}
@
traceLine("I need an include:  <string>!");
insertTextOnce(getFloatingLocation("include files"), "#include
<string>" + endl());
@std::vector<std::string> g() {
...
}
@
traceLine("I need two includes:  <string> and <vector>!");
insertTextOnce(getFloatingLocation("include files"), "#include
<string>" + endl());
insertTextOnce(getFloatingLocation("include files"), "#include
<vector>" + endl());
```

Output:

```
I need an include:  <string>!
I need two includes:  <string> and <vector>!
```

Generated text:

```
#include <string>
#include <vector>

void f(const std::string& s) {
...
}
std::vector<std::string> g() {
...
}
```

See also:

`insertText` 4.6.21, `insertTextOnceToFloatingLocation` 4.6.23, `insertTextToFloatingLocation` 4.6.24, `overwritePortion` 4.6.26, `writeBytes` 4.6.35, `writeText` 4.6.36, `writeTextOnce` 4.6.37

4.6.24 insertTextOnceToFloatingLocation

- procedure **insertTextOnceToFloatingLocation**(location : *string*, text : *string*)

Parameter	Type	Description
location	*string*	the name of a floating location
text	*string*	sequence of characters to insert

Inserts a sequence of characters passed to argument text, at the position of the floating location, whose name is given by the argument location, but only if the text has never been encountered previously by a insertTextOnce() or a writeTextOnce() procedure.

If the floating location doesn't exist, the function raises an error.

In *expansion mode*, the floating location might point into a markup area previously generated. So, on the contrary of insertTextOnce(), the function permits to insert text out of the current markup area.

See also:

insertText 4.6.21, insertTextOnce 4.6.22, insertTextToFloatingLocation 4.6.24, overwritePortion 4.6.26, writeBytes 4.6.35, writeText 4.6.36, writeTextOnce 4.6.37

4.6.25 insertTextToFloatingLocation

- procedure **insertTextToFloatingLocation**(location : *string*, text : *string*)

Parameter	Type	Description
location	*string*	the name of a floating location
text	*string*	sequence of characters to insert

Inserts a sequence of characters passed to argument text, at the position of the floating location, whose name is given by the argument location.

If the floating location doesn't exist, the function raises an error.

In *expansion mode*, the floating location might point into a markup area previously generated. So, on the contrary of insertText(), the function permits to insert text out of the current markup area.

See also:

insertText 4.6.21, insertTextOnce 4.6.22, insertTextOnceToFloatingLocation 4.6.23, overwritePortion 4.6.26, writeBytes 4.6.35, writeText 4.6.36, writeTextOnce 4.6.37

4.6.26 newFloatingLocation

- function **newFloatingLocation**(key : *string*) : *bool*

Parameter	Type	Description
key	*string*	name of a floating position into the output stream

---

Assigns the current position into the output stream to a *floating location* whose name is passed to argument `key`. If the *floating location* already exists, its position is updated.

Example:

```
Roger Rabbit@
newFloatingLocation("Roger Rabbit");
@ doesn't like spinash@
traceLine("Position just after 'Roger Rabbit' is " +
getFloatingLocation("Roger Rabbit"));
```

Output:

```
Position just after 'Roger Rabbit' is 12
```

Generated text:

```
Roger Rabbit doesn't like spinash
```

See also:

```
newFloatingLocation      4.6.25,      getFloatingLocation      4.6.11,
allFloatingLocations     4.6.2,       existFloatingLocation     4.6.9,
getOutputLocation       4.6.16,      removeFloatingLocation    4.6.29,
setFloatingLocation 4.6.32, setOutputLocation 4.6.33
```

4.6.27   overwritePortion

- procedure **overwritePortion**(`location` : *int*, `text` : *string*, `size` : *int*)

Parameter	Type	Description
`location`	*int*	points to a position of the output stream
`text`	*string*	sequence of characters to write
`size`	*int*	size of the portion to overwrite

Writes a sequence of characters passed to argument `text`, at the position of the output stream given by argument `location`. The text overwrites up to `size` characters and inserts the rest if any. The position starts counting at 0.
About the behaviour of the overwriting:

- If there are more than `size` characters in `text`, all *floating locations* that point to `location + size`, or after, are impacted by the insertion of the remaining characters of `text`, and shift for an offset that is worth the size of the text minus the size of the portion to overwrite.

- overwritePortion(*pos*, *text*, 0) is worth insertText(*pos*, *text*).

- If the portion to overwrite is bigger than the length of `text`, all not overwritten characters of the portion are removed.

- overwritePortion(*pos*, `""`, *size*) removes *size* characters from the output stream at position *pos*.

If the position isn't valid, negative or bigger than the end of the output stream, an error is raised.

In *expansion mode*, the position 0 points to the first character written for expansion and cannot exceed the last character written for expansion.

Generally, the position is given by the function getFloatingLocation().

Example:

```
I'll drink a bottle of '@
newFloatingLocation("You'll be drunk!");
@Margaux' year 1994 before smoking a cigar.@
traceLine("Finally, I prefer to drink a bottle of
Saint-Estephe,");
traceLine("I correct the output:");
local iPosition = getFloatingLocation("You'll be drunk!");
overwritePortion(iPosition, "Saint-Estephe", 7 /*size of
'Margaux'*/);
```

Output:

```
Finally, I prefer to drink a bottle of Saint-Estephe,
I correct the output:
```

Generated text:

```
I'll drink a bottle of 'Saint-Estephe' year 1994 before smoking
a cigar.
```

See also:

insertText 4.6.21, insertTextOnce 4.6.22, insertTextOnceToFloatingLocation 4.6.23, insertTextToFloatingLocation 4.6.24, writeBytes 4.6.35, writeText 4.6.36, writeTextOnce 4.6.37

4.6.28   populateProtectedArea

- procedure **populateProtectedArea**(protectedAreaName **:** *string*, content **:** *string*)

Parameter	Type	Description
protectedAreaName	*string*	name of the protected area ; must be unique into the output stream
content	*string*	the content to copy into the protected area

This procedure assigns a content to the protected area whose name is passed to argument protectedAreaName and puts the protected area at the current position of the output stream.

An error is raised if the protected area had already been put into the output stream.

Example:

```
code to generate
@
if !getProtectedArea("reserved for the user")
    populateProtectedArea("reserved for the user", "I can't stand
an empty protected area" + endl());
@I continue the code to generate@
```

Generated text:

```
code to generate
//##protect##"reserved for the user"
I can't stand an empty protected area
//##protect##"reserved for the user"
I continue the code to generate
```

getProtectedArea 4.6.17, getProtectedAreaKeys 4.6.18,
remainingProtectedAreas 4.6.28, removeProtectedArea 4.6.30,
setProtectedArea 4.6.34

4.6.29   remainingProtectedAreas

- function **remainingProtectedAreas**(keys **:** *tree*) **:** *int*

Parameter	Type	Description
keys	*tree*	it will contain keys of all protected areas that haven't been placed into the output stream yet

Looks for all protected areas that haven't been placed into the output stream yet and puts their keys into the well-named argument keys. Both key and value of items are worth the protected area key.

The list is sorted in the lexicographical order rather than the order of entrance.

The function returns how many protected areas are waiting for being put into the output stream. If the variable keys wasn't declared (*local* or *global* or as an attribute of this), samp-1 is returned.

Example - precedent output generated:

```
My first protected area:
//##protect##"hand-typed code"
I don't want to loose this portion of text
after another generation!
//##protect##"hand-typed code"
My second protected area:
//##protect##"don't forget me, please!"
This portion of text is also very important.
//##protect##"don't forget me, please!"
Now, you can erase text, I don't care.
```

Example - the script:

```
My first protected area:
@
local listOfKeys;
local iHowMany = remainingProtectedAreas(listOfKeys);
traceLine("It remains " + iHowMany + " keys to set into the
output file:");
foreach i in listOfKeys traceLine(" '" + i + "'");
traceLine("writing of protected area 'hand-typed code'");
setProtectedArea("hand-typed code");
@My second protected area:
@
set iHowMany = remainingProtectedAreas(listOfKeys);
traceLine("It remains " + iHowMany + " keys to set into the
output file:");
foreach i in listOfKeys traceLine(" '" + i + "'");
traceLine("writing of protected area '" + listOfKeys#front +
"'");
```

```
setProtectedArea(listOfKeys#front);
@Now, you can erase text, I don't care.
@
set iHowMany = remainingProtectedAreas(listOfKeys);
if $iHowMany != 0$ error("shouldn't remain any protected
area!");
traceLine("It doesn't remain any area to set into the output
file");
```

Output:

```
It remains 2 keys to set into the output file:
    'don't forget me, please!'
    'hand-typed code'
writing of protected area 'hand-typed code'
It remains 1 keys to set into the output file:
    'don't forget me, please!'
writing of protected area 'don't forget me, please!'
It doesn't remain any area to set into the output file
```

Generated text:

```
My first protected area:
          //##protect##"hand-typed code"
I don't want to loose this portion of text
after another generation!
//##protect##"hand-typed code"
My second protected area:
          //##protect##"don't forget me, please!"
This portion of text is also very important.
//##protect##"don't forget me, please!"
Now, you can erase text, I don't care.
```

See also:

populateProtectedArea	4.6.27,	getProtectedArea	4.6.17,
getProtectedAreaKeys	4.6.18,	removeProtectedArea	4.6.30,
setProtectedArea 4.6.34			

4.6.30   removeFloatingLocation

- function **removeFloatingLocation**(key : *string*) : *int*

Parameter	Type	Description
key	*string*	name of an existing floating position into the output stream

Removes the *floating location* attached to key and returns its position. If the floating location belongs to a parent output stream, it is removed from the parent.

The function returns a negative integer if the floating location doesn't exist.

See also:

getFloatingLocation	4.6.11,	allFloatingLocations	4.6.2,
existFloatingLocation	4.6.9,	getOutputLocation	4.6.16,

newFloatingLocation 4.6.25, setFloatingLocation 4.6.32, setOutputLocation 4.6.33

4.6.31 removeProtectedArea

- function **removeProtectedArea**(`protectedAreaName` : *string*) : *bool*

Parameter	Type	Description
`protectedAreaName`	*string*	name of the protected area to remove

Removes the protected area whose name is passed to the argument `protectedAreaName` from the list of protected areas that were found into the output file before the generation and that aren't been put into the output stream yet via the `setProtectedArea()` or `populateProtectedArea()` functions.

Example - precedent output generated:

```
My first protected area:
//##protect##"hand-typed code"
I don't want to loose this portion of text
after another generation!
//##protect##"hand-typed code"
My second protected area:
//##protect##"don't forget me, please!"
This portion of text is also very important.
//##protect##"don't forget me, please!"
Now, you can erase text, I don't care.
```

Example - the script:

```
Finally, I have changed my mind and I keep only one protected
area:
@
local listOfKeys;
local iHowMany = remainingProtectedAreas(listOfKeys);
traceLine("It remains " + iHowMany + " keys to set into the
output file:");
foreach i in listOfKeys traceLine(" '" + i + "'");
traceLine("Protected area 'don't forget me, please!'  is
removed");
removeProtectedArea("don't forget me, please!");
set iHowMany = remainingProtectedAreas(listOfKeys);
traceLine("It remains " + iHowMany + " keys to set into the
output file:");
foreach i in listOfKeys traceLine(" '" + i + "'");
traceLine("writing of protected area 'hand-typed code'");
setProtectedArea("hand-typed code");
```

Output:

```
It remains 2 keys to set into the output file:
    'don't forget me, please!'
    'hand-typed code'
```

```
Protected area 'don't forget me, please!'  is removed
It remains 1 keys to set into the output file:
    'hand-typed code'
writing of protected area 'hand-typed code'
```

Generated text:

```
Finally, I have changed my mind and I keep only one protected
area:
          //##protect##"hand-typed code"
I don't want to loose this portion of text
after another generation!
//##protect##"hand-typed code"
```

See also:

```
populateProtectedArea      4.6.27,      getProtectedArea      4.6.17,
getProtectedAreaKeys      4.6.18,    remainingProtectedAreas    4.6.28,
setProtectedArea 4.6.34
```

4.6.32   resizeOutputStream

- procedure **resizeOutputStream**(newSize **:** *int*)

Parameter	Type	Description
newSize	*int*	new size of the output stream

This procedure changes the size of the output stream to newSize. The only allowed resizing is
to reduce the stream (the request is ignored otherwise). If the current position becomes out of the
boundaries, it points to the new end of the output stream.

Example:

```
I write 22 characters.@
traceLine("Current position to the output stream = " +
getOutputLocation());
setOutputLocation(8);
@15@
resizeOutputStream(15);
```

Output:

```
Current position to the output stream = 22
```

Generated text:

```
I write 15 char
```

4.6.33   setFloatingLocation

- procedure **setFloatingLocation**(key **:** *string***,** location **:** *int*)

Parameter	Type	Description
key	*string*	name of a floating position to put into the output stream
location	*int*	the position into the output stream to assign to the key, starting at 0

Assigns a position to the *floating location* whose name is passed to argument `key`.

Example:

```
Roger Rabbit doesn't like spinash@
setFloatingLocation("Roger Rabbit", 5);
traceLine("the floating location 'Roger Rabbit' points just
after 'Roger' = " + getFloatingLocation("Roger Rabbit"));
```

Output:

```
the floating location 'Roger Rabbit' points just after 'Roger' =
5
```

Generated text:

```
Roger Rabbit doesn't like spinash
```

See also:

getFloatingLocation 4.6.11, allFloatingLocations 4.6.2, existFloatingLocation 4.6.9, getOutputLocation 4.6.16, newFloatingLocation 4.6.25, removeFloatingLocation 4.6.29, setOutputLocation 4.6.33

4.6.34 setOutputLocation

- procedure **setOutputLocation**(`location` : *int*)

Parameter	Type	Description
`location`	*int*	points to a position of the output stream

This procedure moves the position of the output stream elsewhere. The position passed to the argument `location` starts at 0.

If `location` is worth **−1**, the cursor moves to the end of the output stream.

Example:

```
I write 22 characters.@
traceLine("Current position to the output stream = " +
getOutputLocation());
setOutputLocation(8);
@one sentence, finally!@
```

Output:

```
Current position to the output stream = 22
```

Generated text:

```
I write one sentence, finally!
```

See also:

getFloatingLocation 4.6.11, allFloatingLocations 4.6.2, existFloatingLocation 4.6.9, getOutputLocation 4.6.16, newFloatingLocation 4.6.25, removeFloatingLocation 4.6.29, setFloatingLocation 4.6.32

4.6.35 setProtectedArea

- procedure **setProtectedArea**(`protectedAreaName` : *string*)

Parameter	Type	Description
`protectedAreaName`	*string*	name of the protected area ; must be unique into the output stream

This procedure puts a protected area at the current position of the output stream, and allows preserving the code of the user between two code generations.

A protected area is bounded by the sequence `##protect##"..."`, put into a comment. The syntax of the comment must conform to the type of the target language expected in the file `outputFileName`. So, an HTML file expects **<!- -** and **- ->**, a JAVA file is waiting for **//** and **"\n"**, ... Don't forget to configure correctly the syntax of comment boundaries with procedures `setCommentBegin()` (see 4.3.181) and `setCommentEnd()` (see 4.3.182).

An error is raised if the protected area had already been put into the output stream.

Example:

```
code to generate
@
setProtectedArea("reserved for the user");
@I continue the code to generate@
```

Generated text:

```
code to generate
//##protect##"reserved for the user"
//##protect##"reserved for the user"
I continue the code to generate
```

See also:

`populateProtectedArea` 4.6.27, `getProtectedArea` 4.6.17, `getProtectedAreaKeys` 4.6.18, `remainingProtectedAreas` 4.6.28, `removeProtectedArea` 4.6.30

4.6.36 writeBytes

- procedure **writeBytes**(`bytes` : *string*)

Parameter	Type	Description
`bytes`	*string*	sequence of bytes to write at the current position of the output stream

Writes a sequence of bytes passed to argument `bytes`, at the current position of the output stream.

A byte is a couple of hexadecimal digits.

Example:

```
codeworker@
writeBytes("4066726565");
@.fr@
```

Generated text:

```
codeworker@free.fr
```

See also:

```
insertText 4.6.21, insertTextOnce 4.6.22, insertTextOnceToFloatingLocation
4.6.23, insertTextToFloatingLocation 4.6.24, overwritePortion 4.6.26,
writeText 4.6.36, writeTextOnce 4.6.37
```

4.6.37 writeText

- procedure **writeText**(`text` : *string*)

Parameter	Type	Description
`text`	*string*	sequence of characters to write at the current position of the output stream

Writes a sequence of characters passed to argument `text`, at the current position of the output stream. It does the same work as the @ tag, but it puts a string into the output stream.

This is the common way to write the symbol '@' into the output stream.

Example:

```
codeworker@
writeText("@");
@free.fr@
```

Generated text:

```
codeworker@free.fr
```

See also:

```
insertText 4.6.21, insertTextOnce 4.6.22, insertTextOnceToFloatingLocation
4.6.23, insertTextToFloatingLocation 4.6.24, overwritePortion 4.6.26,
writeBytes 4.6.35, writeTextOnce 4.6.37
```

4.6.38 writeTextOnce

- procedure **writeTextOnce**(`text` : *string*)

Parameter	Type	Description
`text`	*string*	sequence of characters to write at the current position of the output stream

Writes a sequence of characters passed to argument `text`, at the current position of the output stream, only if the text has never been encountered previously by a `insertTextOnce()` or a `writeTextOnce()` procedure.

Example:

```
@
traceLine("Do you know that Roger Rabbit is tired?");
writeTextOnce("Roger Rabbit is tired");
traceLine("Once again, Roger Rabbit is tired!");
```

```
writeTextOnce("Roger Rabbit is tired");
traceLine("Once more, Roger Rabbit is tired!");
writeTextOnce("Roger Rabbit is tired");
traceLine("The message hasn't been repeated into the generated
text.");
```

Output:

```
Do you know that Roger Rabbit is tired?
Once again, Roger Rabbit is tired!
Once more, Roger Rabbit is tired!
The message hasn't been repeated into the generated text.
```

Generated text:

```
Roger Rabbit is tired
```

See also:

insertText 4.6.21, insertTextOnce 4.6.22, insertTextOnceToFloatingLocation 4.6.23, insertTextToFloatingLocation 4.6.24, overwritePortion 4.6.26, writeBytes 4.6.35, writeText 4.6.36

External bindings

CODEWORKER is written is C++ and works as a standalone console application or as a library. However, it exists some ways to extend CODEWORKER that proposes:

- to link to a JAVA application,

- to declare external functions, developed freely into another library,

5.1 The JAVA binding

A semi-JNI (Java Native Interface) allows catching all messages of CODEWORKER intended to the console. The developer has to develop its own JNI module in C++ to call the object `JNIExternalHandling`.

```
      // file "JNIExternalHandling.h":
 1 /* "CodeWorker":  a scripting language for parsing and
generating text.
 2
 3 Copyright (C) 1996-1997, 1999-2002 Cédric Lemaire
 4
 5 This library is free software; you can redistribute it and/or
 6 modify it under the terms of the GNU Lesser General Public
 7 License as published by the Free Software Foundation; either
 8 version 2.1 of the License, or (at your option) any later
version.
 9
10 This library is distributed in the hope that it will be useful,
11 but WITHOUT ANY WARRANTY; without even the implied warranty of
12 MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU
13 Lesser General Public License for more details.
14
15 You should have received a copy of the GNU Lesser General
Public
16 License along with this library; if not, write to the Free
Software
17 Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA
02111-1307 USA
18
```

```
19  To contact the author:  codeworker@free.fr
20  */
21
22  #ifndef _JNIExternalHandling_h_
23  #define _JNIExternalHandling_h_
24
25  #include <string>
26
27  #ifdef CODEWORKER_JNI
28  # include "jni.h"
29  #else
30      typedef void* JNIEnv;
31      typedef void* jobject;
32      typedef void* jclass;
33      typedef void* jmethodID;
34      typedef void* jobjectArray;
35      typedef void* jstring;
36  #endif
37
38  #include "CGExternalHandling.h"
39
40
41  namespace CodeWorker {
42      class JNIExternalHandling :  public CGExternalHandling {
43      private:
44          JNIEnv& _theEnv;
45          jobject _pJNIObject;
46
47          jclass _pClass;
48          jmethodID _pInputLineMethod;
49          jmethodID _pTraceLineMethod;
50          jmethodID _pTraceTextMethod;
51
52      public:
53          JNIExternalHandling(JNIEnv& theEnv, jobject pJNIObject);
54          virtual ~JNIExternalHandling();
55
56          jstring executeScript(jobjectArray listOfCommands);
57
58          virtual std::string inputKey(bool bEcho);
59          virtual std::string inputLine(bool bEcho);
60          virtual void traceLine(const std::string& sLine);
61          virtual void traceText(const std::string& sText);
62      };
63  }
64
65  #endif
```

LINE 53: the constructor of the C++ class `JNIExternalHandling` that diverts the messages sent to the console by CODEWORKER . It must be called into the function of a C++ JNI module. The two parameters are always passed by JAVA when calling a native interface.

Parameter	Type	Description
theEnv	JNIEnv&	the environment variable for JNI
pJNIObject	jobject	a reference to the JAVA instance whose class name is chosen by the developer freely, and called *mypackage_MyClass* below

The constructor only serves to launch properly `executeScript()` on the instance, and to call functions that catch messages intended to the display.

LINE 56: the method `executeScript()` expects a command line to execute by CODEWORKER and it returns an error message if something failed, or and empty string if success. The method must be called into a C++ JNI module that first calls the constructor seen before:

```
/*
* Class:  mypackage_MyClass
* Method:  myMethod
* Signature:  ([Ljava/lang/String;)Ljava/lang/String;
*
```

Parameter	Type	Description
theEnv	JNIEnv&	the environment variable for JNI
pJNIObject	jobject	a reference to the JAVA instance
listOfCommands	String[]	the command line that CODEWORKER must execute

```
*/
JNIEXPORT jstring JNICALL Java_mypackage_MyClass_myMethod
(JNIEnv *pEnv, jobject pJNIObject, jobjectArray listOfCommands)
{
JNIExternalHandling theExecution(*pEnv, pJNIObject);
return theExecution.executeScript(listOfCommands);
}
```

LINE 59: the callback method `inputLine()` is waiting for a line that the user will put into the standard input, the keyboard generally. The method must be implemented into the JAVA class *mypackage_MyClass* under the prototype: `public String inputLine(boolean bEcho)`, as seen further.

LINE 60: the callback method `traceLine()` passes the message intended to the console to the argument sLine. The method must be implemented into the JAVA class *mypackage_MyClass* under the prototype: `public void traceLine(String sLine)`, as seen further.

LINE 61: the callback method `traceText()` passes the message intended to the console to the argument sText. The method must be implemented into JAVA class *mypackage_MyClass* under the prototype: `public void traceText(String sText)`. An example of source code that illustrates the JAVA part of the JNI may be:

```
package mypackage;
...
public class MyClass {
    ...
    public native String myMethod(String[] listOfCommands);

    public String inputLine(boolean bEcho) {
        byte[] tbLine = new byte[512];
        System.in.read(tbLine);
        return new String(tbLine);
    }
```

```
    public void traceLine(String sLine) {
        System.out.println(sLine);
    }

    public void traceText(String sText) {
        System.out.print(sText);
    }
}
```

Before calling the native method for the first time, don't forget to load your library:
```
try {
    System.loadLibrary("MyLibrary");
} catch(Exception exception) {
    System.out.println("Unable to load the library:  '" +
exception.toString() + "'");
}
```

The following source code is an example of implementation for calling the native method, to put into a method of *MyClass*:
```
String sErrorMessage = null;
try {
    sErrorMessage = myMethod(_tsCommands);
} catch(UnsatisfiedLinkError exception) {
    System.out.println("Unable to link to function
'myMethod(String[])' into the library:  '" + exception.toString()
+ "'");
} catch(Exception exception) {
    System.out.println("Unexpected exception while calling function
'myMethod(String[])' into the library:  '" + exception.toString() +
"'");
}
```

5.2   Developing external functions

We have seen that functions, if not predefined, are implemented by the developer via the script language. But how to populate CODEWORKER with applicative data, or to interact with another libraries?

Let imagine that you have developed a lot of powerful functionalities for your Business and that you want to take advantage of parsing and source code generation processes into your application.

It may be a devoted Server Page, for example: you want to apply templates on your applicative data to display them properly in HTML or anything else and you have integrated a tiny HTTP server. You might implement some external functions that will be called by the template CODEWORKER scripts and that will populate the parse tree with your applicative data, coming from a database or the memory of the application or anywhere.

The prototype of an external function is preceded by the `external` keyword and its body isn't implemented:
```
external-declaration ::= "external" function-prototype ';'
```

An external function conforms to the following C++ `typedefs`:
```
namespace CodeWorker {
```

```
    typedef std::string (*EXTERNAL_FUNCTION)(CppParsingTree_var**);
    typedef std::string (*EXTERNAL_TEMPLATE_DISPATCHER_FUNCTION)(const
std::string&, CppParsingTree_var**);
}
```

where EXTERNAL_FUNCTION represents the general C++ prototype of a classical function or an instantiated template function and EXTERNAL_TEMPLATE_DISPATCHER_FUNCTION represents the C++ prototype of a template function dispatcher and that must be present as soon as at least one instantiated template function exists.

Example:

```
external function f<"x">(i, j:reference, k:variable);
external function f<"y">(i, j:reference, k:variable);
external function g(i, j:reference, k:variable);
```

5.2.1   Doing the C++ binding by hand (good luck!)

The C++ prototype of function f<"x">(i, j:reference, k:variable) conforms to the EXTERNAL_FUNCTION typedef and may be std::string f_x(CppParsingTree_var** tParameters) for example.

As it is an instantiated template function, we need to declare a function whose role is to dispatch dynamically the template functions instantiated on f<...>(i, j:reference, k:variable). Its C++ prototype must conform to the EXTERNAL_TEMPLATE_DISPATCHER_FUNCTION typedef and may be std::string f(const std::string&sTemplateKey, CppParsingTree_var** tParameters) for example. This function must implement the dispatching to the instantiated function corresponding to the *template key*. If the template key is empty, it means that the function must evaluate its own code (f<"">(...) is equivalent to f(...) that is a classical call to a standard function, see 4.2.6).

The C++ prototype of function g(i, j:reference, k:variable) conforms to the EXTERNAL_FUNCTION typedef and may be std::string g(CppParsingTree_var** tParameters) for example.

CODEWORKER exists as a library and provides some C++ headers:

- *"CGRuntime.h"*: allows accessing to all functions and procedures of the interpreter and to the typedefs seen before,

- *"UtlException.h"*: declares the unique type of exception raised by the functions of *"CGRuntime.h"* and called CodeWorker::UtlException,

- *"CppParsingTree.h"*: declares the parse tree, such as:

    - CodeWorker::CppParsingTree_var: declares a variable that points to a parse tree,
    - CodeWorker::CppParsingTree_value: declares a parse tree intended to be pushed to the stack as a local variable; for convenience, a parse tree inherits from CodeWorker::CppParsingTree_var,

External functions must be registered into the library. A unique key represents a function, instantiated or classical, which is the prototype of the function expressed close to the syntax of the scripting language. For example:

- the function `f<"x">(i, j:reference, k:node)` has the following unique key: *"f<x>(i : value, j : reference, k : node)"*,

- the dispatching function `f(i, j:reference, k:variable)` has the following unique key: *"f(i : value, j : reference, k : node)"*,

- the function `g(i, j:reference, k:variable)` has the following unique key: *"g(i : value, j : reference, k : node)"*,

Notice that the template key of the instantiated functions is written without double quotes and that the argument mode is always specified. A space is inserted around the colon that separates the name and the mode of an argument. A space is inserted after the comma that separates two arguments.

To register a classical or an instantiated function, the C++ function `CodeWorker::CGRuntime::registerExternalFunction(...)` must be called:
`CodeWorker::CGRuntime::registerExternalFunction("f<x>(i : value, j : reference, k : node)", f_x);`
`CodeWorker::CGRuntime::registerExternalFunction("g(i : value, j : reference, k : node)", g);`

To register a dispatcher of template functions, the C++ function `CodeWorker::CGRuntime::registerExternalTemplateDispatcherFunction(...)` must be called, but don't forget that this dispatcher function is also a classical function with an empty *template key* (declared as `f_` for example):
`CodeWorker::CGRuntime::registerExternalTemplateDispatcherFunction("f(i : value, j : reference, k : node)", f);`
`CodeWorker::CGRuntime::registerExternalFunction("f(i : value, j : reference, k : node)", f_);`

Now, the body of the function `f`, which takes the role of the template dispatcher, must be implemented.

5.2.2   Generating the C++ binding automatically

It is very tedious to do the C++ binding by hand. Fortunately, CODEWORKER can generate all the C++ binding of external functions automatically. And then, you'll just have to populate the body of the C++ external functions.

The option `-c++external` of the command line asks CODEWORKER for generating the C++ binding. A file name or the radical of a file (without extension) follows the option. CODEWORKER will put together all external functions it has encountered into the *leader script* and all its dependencies (use options `-script` or `-compile` to specify the script to explore recursively). It no error was raised during the execution (option `-script`) or the compilation (option `-compile`), two files are generated:

- The first one, built by taking the radical of the file passed to the command line and adding the extension *".h"*, will describe the declaration of all external functions,

- The second one, built by taking the radical of the file passed to the command line and adding the extension *".cpp"*, will describe the skeleton of all external functions and the implementation of all template dispatchers; you just have to type the user code into the protected areas,

A `init()` function is generated that registers all functions correctly and that must be called explicitly by the user before calling an external function into a CODEWORKER script.

The integrated debugger

The integrated debugger assists in debugging your scripts running. There is no GUI to require and display debug information, just a console.

6.1   Opening the debugger

If you pass option `-debug` on the command line, the debugger will automatically open with the interpreter suspended, and the first line to execute will be displayed. If you want to delimit the debugging session to a part of the project only, type keyword `debug` just before the statement to debug. Generally, the statement is a procedure that interprets a parsing script or a pattern script, such as `generate` or `parseAsBNF`.

6.2   General functionalities

During the execution, the debugger may take the control when:

- a breakpoint in the source code is encountered,

- a step-by-step has been required,

- the developer goes to the next instruction,

- an error is thrown,

Once the debugger is open and the interpreter is suspended, you can work with the program in the following ways:

- inspect the call stack,

- inspect local variables,

- step through functions,

- set and clear breakpoints,

- evaluate some expressions,

The debugger is very convenient during the writing of a parsing script, for example. At any time, it is possible to inspect the parse tree and to understand why it isn't populated as expected.

6.3 Commands of the debugger

All available commands are reported below:

Option	Description
b *filename* at *line* or `breakpoint`	To set a breakpoint in file *filename* at line *line*
c or `clear`	To clear all breakpoints
c *filename*	To clear all breakpoints into file *filename*
c *filename* at *line*	To clear breakpoint located at *line* into file *filename*.
d [*height*]? or `display`	To display the current line the execution has suspended. If the argument *height* was specified, it displays *height* lines before and after the current line. When progressing into a parse script, the current location in the parsed file is also displayed (file name followed by the line/column number).
d *size*	To display the current line the execution has suspended with *size* lines before and *size* lines after.
h or `help` or ?	To display help about commands.
l or `local`	To display local variables present on the stack.
n or `next`	To go to the next statement.
o or `object`	To inspect a variable.
q or `quit`	To quit the debug session.
r or `run`	To continue up to the next suspending cause, breakpoint or error thrown.
s or `step`	To go into function calls or to the next statement.
`stack`	To inspect the call stack.
t or `trace`	To evaluate an expression.

Notice that the debugger never stops on a brace or on a text between '@'. So, breakpoints will have no impact on them, and `step` or `next` commands will ignore them.

Quantifying scripts

7.1 Presentation

CODEWORKER integrates a profiling tool that measures the coverage of scripts you are interpreting and the global time every functions and procedures have run. It helps you quickly and easily find the untested parts of your code and the frequency of passing through code lines. It pinpoints performance bottlenecks quickly. No development time will be wasted tuning code that runs too slow, helping you to develop scripts faster.

Interpreting scripts in quantify mode is about 5% slower than in standard mode.

7.2 Running the profiling tool

The profiling tool may be launched in two ways:

- on the command line with option `-quantify` [*HTML-filename*]`?` to instrument all scripts,

- on a statement into a script. The part of the script that belongs to the statement and all visited scripts (via `generate` or `parseAsBNF` or ...) will be examined. The statement to profile is then prefixed by the statement's modifier *quantify*. For instance:
  ```
  quantify generate("JAVAObject.gen", myClass, myClass.name +
  ".java")
  ```
  The BNF syntax is:
  ```
  quantify ['(' HTML-filename ')']?  statement
  ```

7.3 The profiling results

When the `-quantify` option (or the statement's modifier `quantify`) isn't followed by an HTML file name, the synthetic profiling results are reported to the console:

- each user function appears, recalling the script file where it was defined, and giving how many times it was executed followed by the total execution time in milliseconds,

- each predefined function or procedure appears, giving how many times it was executed,

- the proportion of source code that was executed, considering visited scripts only,

If a file name was specified, the HTML output file highlights all visited script, so as to show parts of the code that are executed a lot and those that are less executed. Each visited line is prefixed by the number of times the controlling sequence has run on it.

Some points to notice:

- the function called `not` represents the unary boolean operator **!**,

- the instruction called `__RAW_TEXT_TO_WRITE` represents the text of a *pattern script* to put into the output stream directly, which is inlayed in **@...@** or in **%>...<%** tags,

- the instruction called `writeText` represents an expression of a *pattern script* that was inlayed in **@...@** or in **<%...%>** tags,

Integrating source code generation into a project

8.1 Reusability

IT people is convinced about the importance of reusability, which is applied to designs, source codes, technical and functional components:

- designs offer to reuse some modeling processes called *design patterns*,

- an object language answers to the concept of reusability with:

  - the polymorphism,
  - the inheritance,
  - the encapsulation,

  Some of them provide the reflectivity, which opens a large field of possibilities, thanks to a kind of generalization of the source code. Some others bring template functions and classes that is another step towards the factorization of the source code.

- components may be seen as a black box intented to provide services, and the way to reuse a component is to embed it into a module (object, library, executable, server) and to offer an interface for accessing its services.

Some points are relative to the concept of reusability, considering the capitalization of skills, the IT knowledge, independently of the Business, but have no satisfying answers today:

- **How to integrate any kind of formal representation as a part of the project's modeling design?**
  A design is built under a modeling tool generally, but the modeling tool cannot accept to enrich the design with a very exotic format that conforms to an unknown syntax and that requires to change the internal representation of the *world* and the way to implement it.

- **How to automatize the implementation of a *design pattern*?**
  Perhaps that, starting from the design, your modeling tool offers the generation of some classical *design patterns*. Perhaps that it is possible, under some constraints, to implement its own *design patterns* via the modeling tool. But what about the flexibility and the readability and the convenience for use and the coverage of wishes?

- **How to automatize the bridge between the design and the implementation?**
  Modeling tools offer to generate the source code corresponding to the design in some classical

languages. The implementation is often restricted to write the skeleton of classes and to implement some *design patterns*. If you want to choose another language:

- you may have to pay fees for a new package,

- you may have to adapt some properties that depend on the target language,

The generated code is quite poor, and you feel that it should be possible to generate more from the design.

- **How to customize the style of the implementation, the features to include?**
  It depends on the flexibility of your modeling tool. The style for implementing some parts (attributes, methods) can be customized often. But changing completely the way an implementation is done may become very tedious and unreadable (if possible!). For example:

  - to move a behaviour out of the class and to put it into a *visitor-like* design pattern where all must be implemented,

  - to implement a wrapper that inserts and updates and extracts objects into/from a database, where all convenient stored procedures are generated too,

  The simplest and the most flexible it appears to change the generation, the furthest you might progress in the complexity and the proposal of new features. For instance, if the chosen syntax for writing JAVA or C++ programs was XML, which is very fashionable for the easiness it is lending itself to a computer processing, the human being wouldn't have been able to write as complex and powerful programs as today, because the syntax isn't adapted and too verbose: too much symbols should be required to express simple concepts.
  If you are convinced of this argument, you will be perhaps convinced that the code generation should be processed thanks to a language as much adapted as possible to simplify its description. It eliminates XML, but also Visual Basic and all *large domain* programming languages (C, C++, JAVA, Fortran, ...).

- **How to preserve a project of the constraining and often definitive choice of a target language**?
  The only way to keep a project independent from the language's choice is to report the details of implementation as much as possible into the design. The most convenient way to express the details of implementation is often to use a programming language, but not systematically. If so, this programming language should be devoted to the phase of modeling of the design and might be translated to a classical programming language.
  Writing to a such language makes rather tricky the validation if no adapted environment is available, so it slows down the development process: changing the design and next, generating code and next, testing and next, taking back corrections to the design and let's continue the loop. If not a lot of protected areas (implementation present in the source code of the target language only, nothing coming from the design) have been populated, a good deal might be to support a set of scripts for each targeted language. The difficulty lies in keeping all sets of scripts at the same level. The inconvenient of supporting multi languages is how to refer to standard libraries: some functionalities exist on a language, some other not, or are exploited differently. However, it is possible to minimize the impact of choosing a language and to isolate rigorously what depends on a programming language exclusively.

- **How to propagate a new feature into a lot of source files?**
  The new feature may consist of serializing all business objects in a XML stream, for instance. It is impossible to implement it with the reflectivity as in JAVA, because one cannot distinguish between an aggregation and a common association, which don't lead to the same serialization (the

---

description of the aggregated object is embedded into the description of the parent). The most convenient way is to dispose of the modeling design and to modify the process of generation as simply as possible.

- **How to limit strongly on the implementation, the impact of modifying the modeling design?** The most information exploitable by the computer you put into the design, the less source code or documentation you will have to modify or to add by hand each time the design will change. The design must be able to express as much concepts as possible, which will have to be implemented automatically.

- **How to automatize the implementation of an architecture?** Another underlying question: how to keep independent of the choice of the architecture while building the core of applications or modules? If you don't dispose of tools for generating automatically the layer of communication with the framework you have chosen:

  - you will do it by hand and waste a lot of time for that,
  - you will exploit some facilities provided by the framework to integrate your implementation, but you will write a framework-dependent source code to help accessing the functional part (see `J2EE` for example).

  In both cases, the required investment for implementing the layer will discourage you to rewrite it later for another framework.

- **How to get back the IT knowledge to another project?** You hope that some developers that own the technical skills will not leave you before the end and that will agree to work on the same technologies.

CODEWORKER proposes an answer for each of these points.

8.2   The interest of controlling the format of the design

The first interest of controlling the format of the design is to be able to acquire data into the source code generator obviously.

The second one is to allow adapting a modeling language to specific needs. It may be to enrich a *UML*[1] design with some features that are necessary to a better mapping to implementation, if you consider that *UML* isn't expressive enough to allow a source code generation as fine as expected and not depending on the target language. Today, thanks to *Rational ROSE*, some more detailed information can be added to an *UML* design, but they are depending on the target language. For instance, if a method's parameter has to be given by reference, the designer has to know that he wants to generate *C++* and then, he writes `std::string&` explicitly as the correct translation. Then, the design cannot be taken out for a *JAVA* code generation after. Finally, there is a strong dependency between the design and the language which will be used for generation.

So, some extensions might be brought on *UML* for remaining free from the choice of the target language:

- containers, such as `list<value-type>` or `hashtable<key-type, value-type>` or `set<value-type>`,

- `PK<basic-type>` for an attribute that holds the role of a kind of *primary key*,

- conditions of existence of an attribute that determine whether an optional attribute must be populated or not,

---

[1]Unified Modeling Language

- checking rules that must be valid if the attribute exists,

- definition of constructors,

- a little *design pattern* called *build*, which is applied to aggregations and that generates a `build<aggregation-name>` method for each constructor of the aggregated object,

- a little *design pattern* called *add*, which is applied to lists of aggregations for building new instances and adding them automatically into the lists,

- `const` and `static` and `virtual` (or `final` keyword, it depends on what is admitted between C++/JAVA philosophy for polymorphism) as specifiers for methods,

- the parameter modes:

    - `in`: parameter is given by reference and can't be changed,

    - `out`: parameter will be created and assigned into the method,

    - `inout`: parameter is given by reference and allows changing,

    - `nothing`: parameter is given by value (copied into the stack in C++),

- the `throw<exception-type>` to specify what kind of exceptions might be raised by a method,

- some *design patterns*:

    - the `visitor`,

    - the redirection of methods and accessors to an encapsulated object,

    - the multi-dispatching like in ADA,

- some free annotations (perhaps not reusable properly), such as `sql`, to specify how to map the design to a SQL schema (attributes to make persistent, objects to map to a table or to merge to another, ...) and to generate the stored procedures for inserting and deleting and selecting objects,

The advantage of enriching *UML* is that one can draw the design into a modeler, under *RATIONAL Rose* for instance, and to put extensions into. It becomes less readable and the project cannot be generated with the modeler anymore, but classes and relationships are built and displayed in a very convenient way. Of course, the design's file must be readable by CODEWORKER and that's the case for *"*.mdl"* files, which are produced by *RATIONAL Rose*[2]. See the **script repository** on `codeworker@free.fr` for taking the adapted parsing script back.

If the readability has suffered too much of adding all these features in a graphical modeler or if the data you want to handle are far from a *UML* representation, you can develop your own modeling language or to adapt one of those proposed into the *script repository*.

Just a point to notice: we don't care about the syntax of the designs to parse, but the structure of the parse tree is very important to warrant the reusability. So, if you want to improve a modeling language provided into the *script repository*, be careful about changes you'll made on the parse tree. Adding new attributes on nodes have no impact on the existing *generation* scripts, but removing or renaming some attributes of the parse tree will change the generated text (some expected attributes will not be found).

So, an effort must be made to document as much as possible the structure of the parse tree, to avoid diverging, so as to allow the reusability of your work or to reuse the work of somebody else.

---

[2]This isn't advertising!

8.3   Driving the implementation with CodeWorker

CODEWORKER provides a scripting language where the syntax is adapted for parsing and code generation tasks. It proposes an easy way to navigate along parse trees too. These three main aspects allow both acquiring data and generating any kind of free text in a very convenient way (adapted syntax and data structures).

CODEWORKER offers some basic functionalities for handling files and directories, which avoid using shell or *Perl* scripts, while building development/test environments within a team for generating/compiling/debugging/sharing source code. The data structure of tree and the `foreach...cascading` statement (see 4.2.5) provide a very convenient way to visualize/navigate along directories.

* TO DO * TO DO * TO DO * TO DO * TO DO ...

Tutorials

Project	Description
JAVA connector	A use-case that demonstrates how to use the JNI interface of CODEWORKER for running it via a JAVA application.
TO DO	A use-case that demonstrates how to exploit the CODEWORKER library inside a C++ application that requires parsing and source code generation.
Server Page	A use-case that demonstrates how to implement external functions for interacting between CODEWORKER and a C++ module (library or application).

INDEX

UUID(), 228

value, 85
variable
    parent, 74
variables
    assignment, 74
    declaring, 72
    navigating, 73
    project, 72
    resolving at runtime, 73
    scope, 72
    this, 72

while, 78
writeBytes(), 291
writefileHook, 90
writeText(), 291
writeTextOnce(), 292