

# Template Parser

= | e

Esito products are copyrighted and all rights are reserved by Esito. This document is also copyrighted and all rights are reserved. This document may not, in whole or in part, be copied, photocopied, translated, or reduced to any electronic medium or machine-readable form without prior consent, in writing, from Esito. The information in this document is subject to change without notice, and Esito assumes no responsibility for any errors that may appear in this document. The references in this document to specific platforms supported are subject to change. Genova, Sysdul, Systemator are trademarks or service marks of Esito.

Java and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries. IBM and Rational Rose are registered trademarks of IBM Corporation. GWT and GWT-based marks are trademarks or registered trademarks of Google Corporation in the U.S. and other countries. Windows, ASP and Visual Basic are registered trademarks of Microsoft Corporation. Enterprise Architect is a registered trademark of Sparx Systems. Any other trademarks or trade names contained herein are the property of their respective owners.

Copyright 2005-2009 © All rights reserved

**Esito AS**  
**Postboks 191**  
**N-1325 Lysaker**  
**Norway**

---

**TEMPLATE PARSER – TOC**

<b>1</b>	<b>Introduction .....</b>	<b>1</b>
<b>2</b>	<b>Parser Setup.....</b>	<b>3</b>
<b>3</b>	<b>Template files.....</b>	<b>5</b>
<b>4</b>	<b>Sections.....</b>	<b>6</b>
<b>5</b>	<b>Iteration statements.....</b>	<b>7</b>
<b>6</b>	<b>Typedefs .....</b>	<b>8</b>
<b>7</b>	<b>Macros .....</b>	<b>9</b>
<b>8</b>	<b>Substitution variables.....</b>	<b>10</b>
<b>9</b>	<b>Template variables .....</b>	<b>13</b>
<b>10</b>	<b>Expressions.....</b>	<b>15</b>
	10.1 Grammar.....	17
<b>11</b>	<b>Loop statements.....</b>	<b>19</b>
<b>12</b>	<b>Conditional statements.....</b>	<b>20</b>
<b>13</b>	<b>Include statement .....</b>	<b>21</b>
<b>14</b>	<b>Context statements .....</b>	<b>22</b>
<b>15</b>	<b>File statements .....</b>	<b>23</b>
<b>16</b>	<b>Copy file statement.....</b>	<b>24</b>
<b>17</b>	<b>Functions .....</b>	<b>25</b>
<b>18</b>	<b>Log statements .....</b>	<b>27</b>
<b>19</b>	<b>Comment statements.....</b>	<b>28</b>
<b>20</b>	<b>Line statements.....</b>	<b>29</b>
<b>21</b>	<b>Indentation statements.....</b>	<b>30</b>
<b>22</b>	<b>Creating a new template set .....</b>	<b>32</b>



## 1 Introduction

This manual describes Genova's Template Parser. The parser is used by Domain Generator, Database Generator, Service Generator and Dialog Generator. The parser is not used by dialog targets generated with Client Generator.

Genova 8.2 introduced generation for user defined targets. Such targets also uses this parser.

In the Genova installation folder (%GENOVA\_HOME%/misc) you will find a Genova Template Editor Eclipse plugin, which is designed to ease the development of templates for the Genova Template Parser. The editor supports syntax highlighting, content assist, and error marking for start- and end statements.

The editor syntax coloring can be customized through the Genova Template Editor preferences. To get content assist proposals from the editor, press Ctrl+SPACE. When there is only one proposal from content assist, the word will auto-complete.

The editor will report errors where a start- or end statement is missing. This includes iteration statements, loop statements, conditional statements, context statements and line statements. These statements are documented further below.

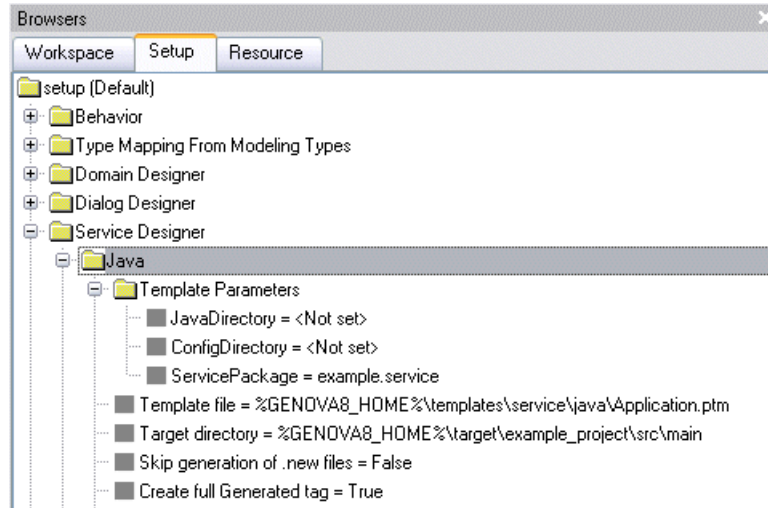
The following chapters contain information about the elements in and use of Template Parser:

- Chapter 1      ["Introduction" on page 1](#) is the chapter you are now reading.
- Chapter 2      ["Parser Setup" on page 3](#) describes the parameters in the setup database used by Template Parser.
- Chapter 3      ["Template files" on page 5](#) explains which templates files the parses uses.
- Chapter 4      ["Sections" on page 6](#) explains the division of the templates into sections.
- Chapter 5      ["Iteration statements" on page 7](#) explains how the parser iterates over the models structures.
- Chapter 7      ["Macros" on page 9](#) explains the usage of macros.
- Chapter 8      ["Substitution variables" on page 10](#) explains how substitution variables works.
- Chapter 9      ["Template variables" on page 13](#) explains how templates variables works.
- Chapter 10     ["Expressions" on page 15](#) describes the grammar of expressions in the parsers language.

- Chapter 11      ["Loop statements" on page 19](#) explains how to create loops in the templates.
- Chapter 12      ["Conditional statements" on page 20](#) explains the how to write conditional statements.
- Chapter 13      ["Include statement" on page 21](#) explains how to include other template files.
- Chapter 14      ["Context statements" on page 22](#) explains how to divide the templates into different context sections.
- Chapter 15      ["File statements" on page 23](#) explains how to specify output files.
- Chapter 16      ["Copy file statement" on page 24](#) explains how maker predefined files part of the generated code.
- Chapter 17      ["Functions" on page 25](#) describes all available functions.
- Chapter 18      ["Log statements" on page 27](#) explains how to write info, error, warning and debug messages from the template files.
- Chapter 19      ["Comment statements" on page 28](#) explains how to write comments in the templates files.
- Chapter 20      ["Line statements" on page 29](#) explains how to concatenate lines.
- Chapter 21      ["Indentation statements" on page 30](#) explains how to handle indentation in both the template files and in the output files.
- Chapter 22      ["Creating a new template set" on page 32](#) explains how to start writing templates for a new target.

## 2 Parser Setup

This chapter gives a general description of the setup parameters used by the different targets based on Template Parser. The description is exemplified with the *Java* target from Service Generator.



**Template Parameters:** This category can hold parameters used in the templates for the target. Parameters defined depend on the target. See ["Template variables" on page 13](#) on how such parameters are used.

**Template file:** This parameter specifies the name of the target's main template file. The template files all have the extension ".ptm".

For more information on template files, see ["Template files" on page 5](#).

**Target directory:** This parameter specifies the default directory where all generated target files are stored.

**Skip generation of .new files:** In the templates there are three different ways to specify an output file, see ["File statements" on page 23](#). By default the *NEWFILE* directive will make the template parser generate a file with extension .new if the file already exists. Setting this setup parameter to *True* will make the template skip the creation of the .new file. No file will be created if the file already exists.

**Create full Generated tag:** When set to *True*, the substitution variable *GeneratedWith* (see ["Template variables" on page 13](#)) will deliver an identification of program version, user doing the generation a timestamp and a version number for the template set. This will produce differences to identical results generated by different users or at a different time. When set to *False*, this replacement string will only contain the text *Generated with 'Genova'* and two results will not show a differences because of a difference in timestamp or name of user doing the generation.

**Give warning for undefined parser variables:** When set to *True*, Template Parser will log a warning message each time an undefined variable is used in the template files. **Note:** This setup parameter is not defined for the Database Generator targets.

**Suppress information messages:** When set to *True*, only a minimum set of information messages are written to the log window during the generation. The next figure shows the log window for a generation when information messages are suppressed.



Both information messages produced by Template Parser itself and messages included in the templates are suppressed, while warning messages are not suppressed. See "[Log statements](#)" on page 27 for more information about the different message types.

### 3 Template files

The template parser uses one main template file as an entry point when a generation is started. The path to and the name of the main template file is specified in the setup parameter *Template File* for each target. Any additional template files used in a generation are reached through include statements from the main template file, see ["Include statement" on page 21](#).

## 4 Sections

A template consists of three sections each starting with a section statement:

- `@reserved`
- `@types`
- `@template`

Only the `@template` section is required, but if present all three sections must be in the main template file. The `@reserved` and `@types` must come before the `@template` section.

Any text before the first section is ignored and may be used to write comments.

The `@reserved` section contains a list of reserved words for the target language, one word per line in the section. Leading spaces are included, trailing spaces are not included.

The reserved words specification is used together with the `%ID:%` usage (see ["Substitution variables" on page 10](#)).

The `@types` section contains conditional type specification, one specification per line. A type specification has the form:

- `type;specification`

A type specification is used inside the `@template` section and may contain parser statements in the same way as the `@template` section itself. A type specification can also be defined in the template section using the `%TYPE-DEF%` directive (see ["Typedefs" on page 8](#)).

A type specification can be used in two contexts, in a `%TYPE:%` usage (see ["Substitution variables" on page 10](#)) and in a `%MACRO:%` usage (see ["Macros" on page 9](#)).

The `@template` section contains the actual template with parser statements and the output text. A parser statement is always surrounded by `'%'` characters. To use `%` in the output text or inside a parser statement one uses two `%` characters, .i.e.:

---

```
%IF:Var="Text with one %% character"%
```

---

## 5 Iteration statements

Each generator based on this parser contains its own hierarchy of nodes of different types, and the parser uses this hierarchy when parsing the templates. For each node one may iterate a typed set of child nodes in the hierarchy. An iteration section is parsed once for each occurrence of the node type. The following statements are used:

- %ITERATE:<NodeType>%
- %BREAK%
- %CONTINUE%
- %ENDITERATE%

The ITERATE: statement starts a new iteration. The <NodeType> is a generator dependent name. The ENDITERATE statement ends current iteration. The block of lines between these two statements is parsed once for each node of the given node type in the hierarchy. Iterations may be nested.

The BREAK statement interrupts the nearest enclosing iteration.

The CONTINUE statement interrupts current pass through the nearest enclosing iteration and the iteration continues with the next pass.

The ENDITERATE statement may an end comment:

- %ENDITERATE:Comment text%

Inside iterations one may test if the iteration is the first or the last one:

- %IF:IsFirst%
- %IF:IsLast%

The IsFirst and IsLast tests both references to the nearest enclosing iteration. See "[Loop statements](#)" on page 19.

## 6 Typedefs

A *typedef* statement defines a new type which can be used both in a %TYPE:% statement (see ["Substitution variables" on page 10](#)) and as a macro in a %MACRO:% statement (see ["Macros" on page 9](#)). Defining a type using a *typedef* statement is the same as defining the type in the @type section of the main template file (see ["Sections" on page 6](#)).

- %TYPEDEF% type;specification

The specification ends at the end of the line containing the *typedef* statement.

A specification can contain any parser directives. The specification is not parsed at the point of definition. It is parsed each time the type is used in the templates and then in the context of that usage.

A type defined in a *typedef* statement is available from the first time the statement is read, but only on the condition that the definition is in a template part that is included as a part of the output. I.e. a definition in a *false* part of a conditional statement will not define a new type (see ["Conditional statements" on page 20](#)). Neither will a definition in an iteration statement (see ["Iteration statements" on page 7](#)) or a loop statement (see ["Loop statements" on page 19](#)) come into usage if the iteration or loop is traversed zero times.

Redefinition of a type will result in a warning from the parser. However, defining a type more than once with the same content is not regarded as a redefinition and will not give a warning.

## 7 Macros

The definitions in the types section (see ["Sections" on page 6](#)) may, in addition to be used as a type conversion (see ["Substitution variables" on page 10](#)), also be used as macros. A type specification consists of a name and a type content separated by a ";":

- type;specification

The specification is used as a macro in the template section with a macro statement:

- %MACRO:type%

When the parser finds a macro statement inside the template section it searches the types section for the type and if found replaces the macro statement with the type specification. The specification may contain parser statements in the same way as the @template section itself and macros may use other macros.

## 8 Substitution variables

Each node contains substitution variables specific for a generator. When used in a template the variable is replaced with a string value delivered by the parser. These variables can be used in the templates in three different ways:

- As a variable in an expression, see ["Expressions" on page 15](#)
- %ID:<Name>%
- %TYPE:<Name>%

In the form of a variable in an expression a regular substitution with the variables value is performed.

The %ID:<Name>% also makes a regular substitution, but the result is tested against the reserved words specified in the @reserved section. Any match results in an error message.

Like the other two, the %TYPE:<Name>% first makes a regular substitution. Then the result is used to find a type specification from the @types section. If the type is found, the type specification is used. If it is not found, an error message is produced.

A type specification may contain parser statements and will be parsed in the same way as the lines in the template file.

The name of a substitution variable consists of an optional node type and the name of the variable. A node type specifies the node in the iteration structure, see ["Iteration statements" on page 7](#). I.e.:

Class.Name references to a substitution variable Name defined at "Class" node when inside an %ITERATE:Class% iteration.

Two predefined node types are known by the parser, "Parent" and "TopParent". The predefined node type "Parent" always references to the iteration one level about current level, while "TopParent" references to the outermost iteration level. In the code below

---

```
%ITERATE:Class%
%ITERATE:Attribute%
%Class.Name%.%Name%
%Parent.Name%.%Name%
%ENDITERATE%
%ENDITERATE%
```

---

both line 3 and line 4 will give the same result:

"ClassName.AttributeName"

The node type may be a sequence of types, i.e. Parent.Parent or Class.Parent.

When the parser searches for a substitution variable it starts by searching the predefined variables:

Name	Description
Count<NodeType>	Number of child nodes of type NodeType. If the iterating over the same node types multiple times the number for the nearest enclosing iteration is returned.
NodeType	Node type of the nearest enclosing iteration.
Today	Current date and time: "2006.11.23 14:25:43".
Version	Current Genova version: "820A".
UserName	Name of current user: "User".
GeneratedWith	String identifying Genova version, user and current time: "Generated with 'Genova' version 8.3.0.A by User on 2009.04.23 14:25:43 Template version 8.3.0.A. May 04, 2009".
TemplateVersion	The template version part of GeneratedWith.
TemplateDirectory	Template directory for the generation as specified in the Setup database: "C:\Genova8\template\service\java".
TargetDirectory	Target directory for the generation as specified in the Setup database: "C:\Genova8\target\service\java".
SkipNewFiles	<i>True</i> if the "Skip generation of .new files" flag in the setup database is set to <i>True</i> for the target.

If no match is found, the parser continues by asking current node and, if no match there either, the search continues to the node's parent node. This is done recursively until either a match is found or the top is reached, i.e. the search follows the iteration hierarchy.

If no match is found in any nodes, the template parameters for the generation target in the setup database are searched. If no match there too, an empty string is returned.

Substitution variables available for each node type are specified for each different generator. The node types reflect a Genova model element and for all nodes the properties of the corresponding model element are available as a substitution variable. I.e. when iterating over classes a Class node has all the properties of Class available such as Name, Description Logical package etc. as substitution variables. For model elements defined in the UML model, user defined tagged values are also available as properties and then also available as substitution variables. How to incorporate your own tagged values in Genova is described in [section 8.5 on page 60](#) in the User Guide manual.

Property names may contain spaces while space is not allowed in a variable name. The parser converts all property names to legal variable name by removing spaces and uppercasing each letter after a space. The table below shows the available variables for a Class, both as named in the classes property dialog and as substitution variables.

<b>Property name</b>	<b>Substitution variable name</b>
Name	Name
Description	Description
Logical package	LogicalPackage
Component package	ComponentPackage
Visibility	Visibility
Abstract	Abstract
Persistent	Persistent
Title	Title

This schema for substitution variables makes it possible to find many of the available substitution variables by looking at the properties for a model element. But other variables are defined for the different node types and a description of such variables are found in the description of the node types in the different generators.

## 9 Template variables

The templates read by the parser may define and use variables. The template variables syntax is:

- `%NodeType#VariableName%`

A variable is defined at an iteration level and the node type part specifies the iteration level, see "[Iteration statements](#)" on page 7. A variable is never declared, it comes into existence when first given a value with the SET statement:

- `%SET:TemplateVariable=Expression%`

The scope for a variable is on the definition level and all iteration levels inner to the definition level.

The following example illustrates the usage of variables

---

```
1. %SET:ClassModel#Counter=0%
2. %ITERATE:Class%
3. %SET:Parent#Counter=ClassModel#Counter+1%
4. %SET:Class#EntityName=Name++Parent#Counter%
5. %ITERATE:Attribute%
6. %SET:Attribute#Name=Class#EntityName++"."++Name%
7. %ENDITERATE%
8. %ENDITERATE%
```

---

This structure defines three variables, one on each level of iteration. This example is from the ClassModel generation and the outermost node type in this generator is ClassModel.

Line 1: Defines a new variable ClassModel#Counter and sets the variable value to 0. The variable could at this level also be named without using the node type, i.e. #Counter. A variable always contains the # but without any node type the variable is expected to be at current iteration level.

Line 2: Iterate through all classes, the node type is Class.

Line 3: Changes the variable defined in line 1. The statement will increase the variable with one. In this line we see two different ways of identifying a variables node level, either relative by using the Parent specification or by using the node type itself, in this case ClassModel.

Line 4: Defines a new variable EntityName at the Class level and sets its value to a string consisting of the name of the class (using the substitution variable Name) concatenated with the value of the Counter variable.

Line 5: Iterate through all attributes contained in current class. The node type is Attribute.

Line 6: Defines a new variable `Attribute#Name` and assign it a string value consisting of the string in the variable `Class#EntityName` and the name of the attribute separated by a `'.'`.

Line 7: Ends the `Attribute` iteration. The variable `Attribute#Name` goes out of scope.

Line 8: Ends the `Class` iteration. The variable `Class#EntityName` goes out of scope.

This example shows single variables, but variables may be indexed:

- `%SET:VariableName[Expression]=Expression%`

Changing the example above will illustrate the usage:

---

```

1. %SET:ClassModel#Counter=0%
2. %ITERATE:Class%
3. %SET:Parent#Counter=ClassModel#Counter+1%
4. %SET:ClassModel#EntityName[Parent#Counter]=Name++Parent#Counter%
5. %ITERATE:Attribute%
6. %SET:Attribute#Name=ClassModel#EntityName[ClassModel#Counter]++"."++Name%
7. %ENDITERATE%
8. %ENDITERATE%
9. %SET:#Index=1%
10. %LOOP:ClassModel#Counter%
11. %ClassModel#EntityName[#Index]%
12. %SET:#Index=#Index+1%
13. %ENDLOOP%

```

---

Line 4: This line now defines an array of variables on the `ClassModel` level.

Line 6: This `SET` definition uses the correct array element and will assign the same value to the `Attribute#Name` variable as in the first example.

Line 8: In this example the `EntityName` array is defined at the `ClassModel` level and the array does not go out of scope when we reach this line.

Line 9 to 13 shows a loop accessing each of the elements in the `EntityName` array.

The example shows an array indexed with numbers, but the index may not be a number, the index can be any string. If we change line 4 to:

---

```
%SET:ClassModel#EntityName[Name]=Name++Parent#Counter%
```

---

The index values will be the name of each class.

One may use multiple indexes, i.e:

---

```
%SET:ClassModel#AttributeName[Class.Name][Attribute.Name]=Class.Name++"."++Attribute.Name
```

---

## 10 Expressions

The parser supports boolean, arithmetic and string expressions. All expression types are strings, but boolean expressions evaluates to strings that contains one of the two string values "True" and "False", while an arithmetic expression evaluates to a string containing an integer, i.e "-45" og "1267". A string expression evaluates to any string.

In all the different expressions parentheses "(" and ")" can be used to group the expression.

### String constants:

A string constant is written inside double quotes, i.e.:

---

```
"This is a string constant"
```

---

To use double quotes in a string write is a two double quotes, i.e.:

---

```
"This is a ""string"" constant with quotes around the word string"
```

---

### Conditional expression:

A conditional expression are written with the ternary operator "?:" and provides a way to make a choice between two values. I.e.:

---

```
BooleanExpression ? Expression1 : Expression2
```

---

will if the boolean expression evaluates to "True" give Expression1 as result, but if the boolean expression evaluates to "False" the result will be Expression2.

### Boolean expression:

A boolean expression are written with the unary operator "!" (not) and the two binary operators "&" (and) and "|" (or) and evaluates to a boolean value. I.e.:

---

```
(BoolValue1 & !BoolValue) | BoolValue3
```

---

The three operator follow the precedence sequence "!", "&", "|". I.e.:

---

```
BoolValue1 | BoolValue2 & BoolValue3
```

---

is the same as

---

```
BoolValue1 | ( BoolValue2 & BoolValue3)
```

---

When a boolean expression is evaluated operators with same precedence are evaluated left to right. The evaluation stops when the outcome of the expression is unambiguous, i.e.:

---

```
"False" & BoolValue
```

---

will never evaluate BoolValue while the reversed expression

---

```
BoolValue & "False"
```

---

always will evaluate the BoolValue.

### **String expression:**

A string expression are written with the binary operator "++" (string concatenation). I.e.:

---

```
"String1" ++ "String2"
```

---

evaluates to the string

---

```
"String1String2".
```

---

### **Relational expression:**

A relational expression are written with one of the six binary relational operators "<" (less), "<=" (less or equal), "==" (equal), "!=" (not equal), ">=" (greater or equal) and ">" (greater). I.e.:

---

```
Variable < 24
```

---

The operands are string, but the comparison used depends of the values. If both values are numbers then numeric comparison is used, in any other case a string comparison is used.

If there is a possibility that both operators are numbers but the desired comparison is a string comparison, one may achieve this by concatenating both operands to the same string.

The relation

---

```
"2" < "10"
```

---

will use numeric comparison and evaluate to "True", while

---

```
"a"++"2" < "a" ++ "10"
```

---

will use string comparison between the two strings "a2" and "a10" and evaluate to "False", which is the same result as one will get if one do a string comparison between "2" and "10".

The result of a relational expression is a boolean value.

**Arithmetic expression:**

An arithmetic expression are written with one of the six binary arithmetic operators "+" (plus), "-" (minus), "\*" (multiplication), "/" (division), "~" (modulus), AND "^" (power).

The arithmetic does not support decimal numbers, all calculations are integer based.

The operators are divided into three precedence groups:

- "+", "-"
- "\*", "/" "~"
- "^"

Operators with equal precedence are evaluated left to right.

**Functions:**

A function consists of a name and a comma separated list of parameters enclosed in parenthesis. I.e.:

- UPPER("string")

The functions known to the parser are found in chapter ["Functions" on page 25](#)

**10.1 Grammar.**

This section contains the full grammar of expressions.

```

Expression ::= CondExpression
CondExpression ::= BoolExpression
CondExpression ::= BoolExpression ? CondExpression : CondExpression
BoolExpression ::= BoolTerm
BoolExpression ::= BoolExpression | BoolTerm
BoolTerm ::= BoolFactor
BoolTerm ::= BoolTerm & BoolFactor
BoolFactor ::= BoolPrimary
BoolFactor ::= !BoolFactor
BoolPrimary ::= RelExpression
RelExpression ::= StringExpression
RelExpression ::= StringExpression < StringExpression
RelExpression ::= StringExpression <= StringExpression
RelExpression ::= StringExpression > StringExpression
RelExpression ::= StringExpression >= StringExpression
RelExpression ::= StringExpression = StringExpression
    
```

$\text{RelExpression} ::= \text{StringExpression} \neq \text{StringExpression}$   
 $\text{StringExpression} ::= \text{AritExpression}$   
 $\text{StringExpression} ::= \text{StringExpression} ++ \text{AritExpression}$   
 $\text{AritExpression} ::= \text{AritTerm}$   
 $\text{AritExpression} ::= + \text{AritTerm}$   
 $\text{AritExpression} ::= - \text{AritTerm}$   
 $\text{AritExpression} ::= \text{AritExpression} + \text{AritTerm}$   
 $\text{AritExpression} ::= \text{AritExpression} - \text{AritTerm}$   
 $\text{AritTerm} ::= \text{AritFactor}$   
 $\text{AritTerm} ::= \text{AritTerm} * \text{AritFactor}$   
 $\text{AritTerm} ::= \text{AritTerm} / \text{AritFactor}$   
 $\text{AritTerm} ::= \text{AritTerm} \sim \text{AritFactor}$   
 $\text{AritFactor} ::= \text{AritPrimary}$   
 $\text{AritFactor} ::= \text{AritFactor} ^ \text{AritPrimary}$   
 $\text{AritPrimary} ::= ( \text{CondExpression} )$   
 $\text{AritPrimary} ::= \text{Primary}$   
 $\text{Primary} ::= \text{Function}$   
 $\text{Primary} ::= \text{Variable}$   
 $\text{Primary} ::= \text{Number}$   
 $\text{Primary} ::= \text{String}$   
 $\text{Function} ::= \text{FunctionName}()$   
 $\text{Function} ::= \text{FunctionName}(\text{Parameters})$   
 $\text{FunctionName} ::= \text{Name of a predefined function}$   
 $\text{Parameters} ::= \text{CondExpression}$   
 $\text{Parameters} ::= \text{Parameters}, \text{CondExpression}$   
 $\text{Variable} ::= \text{TemplateParameter}$   
 $\text{Variable} ::= \text{SubstitutionVariable}$   
 $\text{Variable} ::= \text{TemplateVariable}$   
 $\text{TemplateParameter} ::= \text{Name of template parameter in the setup database.}$   
 $\text{SubstitutionVariable} ::= \text{VariableName}$   
 $\text{VariableName} ::= \text{Name of a property for an iteration node}$   
 $\text{VariableName} ::= \text{NodeName}.\text{VariableName}$   
 $\text{TemplateVariable} ::= \text{NodeName}\#\text{ArrayName}$   
 $\text{ArrayName} ::= \text{Name}$   
 $\text{ArrayName} ::= \text{Name} \text{ArrayIndex}$   
 $\text{ArrayIndex} ::= [\text{CondExpression}]$   
 $\text{ArrayIndex} ::= \text{ArrayIndex}[\text{CondExpression}]$   
 $\text{NodeName} ::= \text{Name of an iteration structure}$   
 $\text{String} ::= \text{A sequence of any characters inside double quotes ("). A " inside a string are written as two quotes ("").}$   
 $\text{Number} ::= \text{Any sequence of integers}$   
 $\text{Name} ::= \text{Any non quoted string which is not a number and don't contain any of the operators used in this grammar}$

## 11 Loop statements

The loop statement executes a loop a given number of times.

- `%LOOP:ArithmeticExpression%`
- `%BREAK%`
- `%CONTINUE%`
- `%ENDLOOP%`

The `LOOP:` statement starts a new loop. The arithmetic expression is evaluated and the loop is then executed specified number of times. If the expression evaluates to a negative value the loop is parsed zero times. The `ENDLOOP` statement ends current loop. Loops may be nested.

While iterations have a type and define scope for variables, loops do not change the variable scope.

The `BREAK` statement interrupts the nearest enclosing loop.

The `CONTINUE` statement interrupts current pass through the nearest enclosing loop and the loop continues with the next pass.

The `ENDLOOP` statement may an end comment:

- `%ENDLOOP:Comment text%`

The nesting of iterations creates a stack of node types. Two special loops are defined that makes it possible to loop through the stack of iterations:

- `%LOOP:Parent%`
- `%LOOP:ParentReverse%`

**Parent** starts with current nodes parent and goes to the top while **ParentRevers** starts at the top and goes down to current node. Current node is not included in the loop.

While looping the stack, template and substitution variables available are as if that node current in the loop is current in the iteration stack. You may access variables from current node and above in the stack, but you may not access variables only visible from the child nodes.

## 12 Conditional statements

The parser recognizes simple conditional statements with expressions:

- `%IF:<BooleanExpression>%`
- `%ELSEIF:<BooleanExpression>%`
- `%ELSE%`
- `%ENDIF%`

A conditional statement consist of an IF part followed by any number of ELSEIF parts, then an optional ELSE part and an ending ENDIF.

A boolean expression (see ["Expressions" on page 15](#)) is either "True" or "False". Only one part of a conditional statement will be selected, the one first evaluating to "True". The ELSE part will only be selected if all IF and ELSEIF conditions evaluate to "False".

The ELSE and ENDIF statements may contain an end comment:

- `%ELSE:Comment text%`
- `%ENDIF:Comment text%`

Conditional statements may be nested.

## 13 Include statement

The include statement is used to read other template files than the main file.

- `%INCLUDE% <templatefile>`

Include the template file. When the included template file has been parsed, the parser continues with the line after the include statement.

The template file name can either contain an absolute path or it may contain a relative path. In the second case the location will be relative to the location of the file containing the include statement.

## 14 Context statements

A context is a template defined boolean variable that may be set and tested with a conditional statement.

- `%CONTEXT:<context>%`
- `%ENDCONTEXT%`

Inside the context the variable is True, else it is false. The conditional statement has the form:

- `%IFCONTEXT:context%`
- `%ELSE%`
- `%ENDIF%`

Contexts may be nested and if inside more than one context, the IF test will give a True value for all of them.

## 15 File statements

The file statements are used to specify the output files used by the parser. It recognizes four different file statements:

- %FILE% <filename>
- %NEWFILE% <filename>
- %IGNOREFILE% <filename>
- %ENDFILE%

The three first statements open a file for writing, while the last statement closes current output file. The three different file open statements is used to specify how the parser should treat already existing files.

**FILE:** If the file exists it will be overwritten.

**IGNOREFILE:** If the file exists the intended output will be discarded.

**NEWFILE:** If the file exists it will not be overwritten. The output will be written to a file with the same name, but with an additional extension `.new`. This behavior can be overridden from the setup database with the "Skip generation of `.new` files" flag for the target in the setup database. If the flag is defined and set to "True", the **NEWFILE** statements will behave in the same way as the **IGNOREFILE** statements, i.e. the output will be discarded if the file exists.

When the parser is writing to one output file you may change to another file with the use of one of the three statements opening an output file.

**ENDFILE:** The current output file will be closed. If more than one outfile was open the output continues with the previous file as current outputfile. The file open statements together with the **ENDFILE** statement behave as a stack.

**Note:** Any output parsed when no output file is open will be discarded.

The parser language has two statements which may alter the sequence of the files on the file stack:

- %PUSHFILE% <filename>
- %POPFIL%

The **PUSHFILE** statement takes a filename as input parameter. This file must already exist on the stack, i.e. it must have been opened with one of the file open statements. The **PUSHFILE** statement takes the file from its current position and put it on the top of stack making it the current output file.

The **POPFIL** statement takes current file (the one on top of the stack) and puts it at the bottom making the new top file the current output file.

## 16 Copy file statement

In some cases a template set may contain files intended to be part of the generated target. Examples of such files are bitmaps distributed with a template set. The copy file statement may then be used in the templates to copy the file to the target directory. The statement has the form

- `%COPYFILE% source file | destination file`

Both the source file and the destination file name may contain an absolute or a relative path. If a relative path is used the source name is regarded as relative to the location of the file containing the COPYFILE statement. In the destination name it is regarded as relative to the target directory as specified in the setup database.

## 17 Functions

The parser knows and uses a number of functions. All these functions return a string. Most of the functions is used to convert an input string, but some utility functions are also defined.

The string functions take an expression as parameter. The expression is evaluated and then the string function is used on the resulting value.

The parser knows the following string functions:

Name	Description
UPPER	Uppercase all of the result.
LOWER	Lowercase all of the result.
NOWS	Remove whitespaces, ' ' and tabulator.
NOSEP	Remove separators, '/', '\ and '.'
NOSEP_CAMEL	Remove separators, '/', '\ and '.'. Return camel cased string.
UNCAMELIZE	Remove camel casing from string, insert '_' instead.
SLASH_TO_DOT	Convert '/' or '\' to '.'.
DOT_TO_SLASH	Convert '.' to '/'.
DOT_TO_UNDER	Convert '.' to '_'.
SPACE_TO_UNDER	Convert ' ' to '_'.
BSLASH_TO_SLASH	Convert '\' to '/'.
SHORT	Take first substring using '/', '\ or '.' as end marker.
FIRST_UPPER	Uppercase first character.
FIRST_LOWER	Lowercase first character.
HTML	Replace HTML reserved characters with their HTML escape sequence.
JAVADOC	Expand each line in a multiline result with a leading " *".
EXPAND_ENV	Expand any environment variables in the result.

In addition to the string functions the parser knows four boolean functions:

Name	Description
EXISTS	Check if a variable exists. The result is "False" if the variable is an undefined substitution variable, a substitution variable with undefined value (seen as <Not set> in the browser) or if the variable is a template variable and has not been given a value. In all other cases the function returns the value "True".

---

<b>Name</b>	<b>Description</b>
DEFINED	Check if a variable is defined. The result is "False" if the variable is undefined or if exists but has the value "False" or the value "" (empty string). In all other cases the function returns the value "True".
ISNUMBER	Check if an expression evaluates to a number.
ISBOOLEAN	Check if an expression evaluates to a boolean value.

## 18 Log statements

The template language supports four different types of log messages:

- %INFO:Expression%
- %WARNING:Expression%
- %ERROR:Expression%
- %DEBUG:Expression%

All four statements send a message to the Genova log. The message is a parser expression and may contain a mixture of variables and text constants. If one wants to output a pure text without the use of any variable values the text must be written as an expression text constant, i.e. the text must be written inside double quotes ("). To create complex messages use the string concatenation operator ++.

**INFO:** The value of the expression is written to the log as an information message without influencing the generation.

**WARNING:** The value of the expression is written to the log as a warning message together with the name and line number of the template file holding the warning statement. The warning message does not influence the generation.

**ERROR:** The value of the expression is written as an error message together with the name and line number of the template file holding the error statement. The error message shows up as a message box and when the user clicks the OK button, the generation is aborted.

**DEBUG:** In debug mode the value of the expression is written as a debug message together with the name and line number of the template file holding the debug statement. If the parser is not in debug mode, the statement is ignored. A generation is in debug mode if the target definition in the set-up database contains the variable **Debug** and this variable has the value "True". The variable should be defined on the same level as the **Template Directory** variable for a target and not as a template parameter.

## 19 Comment statements

The comment statement is used to make comments inside the templates. The statement has the form:

- `%REM:Comment Text%`

If the ending % is omitted, the rest of the line is treated as a comment.

Comment statements may only be used inside the template section, see ["Sections" on page 6](#) about the different sections in the templates. But since all text before the first section is ignored by the parser, that part of the main template file may be used to write any comments giving an overall description for a set of templates.

## 20 Line statements

The main purpose of line statements is to make template files more readable. All output inside a line directive will be written to the same line even if the template contains more than one line.

- %LINE%
- %ENDLINE%

## 21 Indentation statements

Indentation is used to increase the readability of structured languages. The templates use a structured language and to make the templates easy to read, we want to use an indentation that follows the structure of the template statements. But the target languages are also often structured and we also want the results to have an indentation that makes the generated code easy to read. However, in the templates we have a mixture of these two languages, the templates language and the target language, and this makes it difficult to write templates that support well an easy indentation for both templates and target text.

The parser supports three different forms of indentation. By default, the parser ignores leading whitespaces in the templates and all output lines will be without leading spaces. If the target text does not need any indentation you may let the indentation in the templates follow the structure of the template language.

The two different indentation statements are used to manage the indentation in other ways than the default.

The first form makes all whitespaces significant and is managed with the two statements:

- `%INDENT%`
- `%ENDINDENT%`

The indentation statement will make the parser preserve any leading whitespace characters and thus preserve the indentation used in the template files when writing the target code. When using this form, any template statements are written without leading spaces, while the target text is written with the indentation preferred in the output. This form is preferred if the target text uses a lot of indentation while the structure from the template language are simple.

The third form manages the indentation by using an indentation statement to increase or decrease the indentation used in the output. The statement takes a number as parameter, defining the change in indentation.

- `%INDENT:Number%`

A positive number increases the indentation by that number of spaces, while a negative number decreases the indentation.

When this form of indentation is used the parser ignores any leading whitespaces, and the templates can be written with an indentation making

it easy to read the templates, while the use of the `INDENT` statement make sure the output also gets a desired indentation.

The number in the `INDENT` statement could actually be an expression and as such the indentation could be controlled by variables or setup parameters.

## 22 Creating a new template set

The user may create new generator targets in the setup database, see [section 8.11 on page 66](#) in the User Guide. To be able to use the new target for a generator, a template set is needed. This template set may have one or more template files as explained in [Chapter 3 "Template files" on page 5](#) and [Chapter 13 "Include statement" on page 21](#).

The main template file must be structured according to the description in [Chapter 4 "Sections" on page 6](#). The name and location of the main template file must match the *Template File* parameter for the given setup target.