Grammar for IDL Template Files

An IDL template file contains the rules that the Software AG IDL Compiler uses - together with the IDL file - to generate interface objects, skeletons and wrappers for a programming language. The Developer's Kit provides several templates for various programming languages.

Warning:
The information in this section is intended for users who wish to write their own template files. Do not change the delivered template files.

This document explains the syntax of the template files in a formal notation. For an introduction on how to write template files, see *Writing Template Files for Software AG IDL Compiler*.

Software AG Template File Grammar

Syntax

```
{ statement }
```

Description

A template contains the rules which the IDL Compiler uses with the IDL file. The template is the lexical entity to start with.

Example

See under the lexical entity *`statement`*.

assign_statement

Syntax

```
assign_string_statement | assign_integer_statement
```

Description

These statements are used

- to assign strings to *`variable_of_type_string`* and *`variable_of_type_indexed_string`*,
- to compute values and assign them to *`variable_of_type_integer`*.

Example

See the lexical entities *`assign_string_statement`* or *`assign_integer_statement`*.

assign_integer_statement

Syntax

```
%compute variable_of_type_integer string_with_expression_contents
```

Description

Compute the expression in *`string_with_expression_contents`* and assign the result to *`variable_of_type_integer`*.

Example

```
%compute a "&b * (%before + %after) / %eLength"
```

assign_string_statement

Syntax

```
%assign variable_of_type_string string |
%assign variable_of_type_indexed_string string
```

Description

Assign the string contents to *`variable_of_type_string`* or *`variable_of_type_indexed_string`*

Example

```
%assign A "Assign this string to variable A" %assign A[5] "Assign this
        string to occurrence 5 of variable A"
```

block

Syntax

```
'{' statement [ block ] '}'
```

Description

A block is a sequence of statements.

Example

See the lexical entity *`statement`*.

compare_expression

Syntax

```
compare_strings [logical_compare_operator compare_strings]
```

Description

The *logical_compare_operator* performs a logical operation of two compare_strings. See the description of `logical_compare_operator` and `compare_strings` for specific information.

Example

```
%if "&i" > "3" && "&k" < "20"
{
"the variable i is greater than three AND the variable k is less than twenty"
}
%if "&i" = "1" || "&i" > "3"
{
"the variable i is one OR is greater than three"
}
```

compare_strings

Syntax

```
string [ compare_operator ] string
```

Description

Compare two strings for a logical condition. The condition can be TRUE or FALSE.

Example

See lexical entity *compare_operator*.

compare_operator

Syntax

Operator	Meaning
	equal to(default)
=	equal to
<>	not equal to
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to

Description

The logical operators are used in the lexical entity *compare_strings*.

Example

equal to (default)	`"A" "?A"`
equal to	`"A" = "?A"`
not equal to	`"A" <> "?A"`
less than	`"A" < "?A"`
less than or equal to	`"A" <= "?A"`
greater than	`"A" > "?A"`
greater than or equal to	`"A" >= "?A"`

control_statement

Syntax

```
assign_statement | definition-statement | file_handling_statement |  if_statement  | loop_statement | output_control_statement
```

Description

control_statements determine the processing logic of a template. They do not create output.

Example

See lexical entities:

- *assign_statement*
- *definition-statement*

- *file_handling_statement*
- *if_statement*
- *loop_statement*
- *output_control_statement*

definition-statement

Syntax

```
definition-of-base-type-template |
definition-of-direction-template |
definition-of-group-template |
definition-of-index-template |
definition-of-line-number-format-template |
definition-of-member-separator-template |
definition-of-names-format-template |
definition-of-nest-level-format-template |
definition-of-parent-identifier-template |
definition-of-parent-index-template |
definition-of-structure-template |
definition-of-UnboundedArray-template.
```

Description

`definition_statements` give directives to the IDL Compiler. They do not create output.

Example

See lexical entities:

- *definition-of-base-type-template*
- *definition-of-direction-template*
- *definition-of-group-template*
- *definition-of-index-template*
- *definition-of-line-number-format-template*
- *definition-of-member-separator-template*
- *definition-of-names-format-template*
- *definition-of-nest-level-format-template*
- *definition-of-parent-identifier-template*
- *definition-of-parent-index-template*

- *definition-of-structure-template*
- *definition-of-UnboundedArray-template*

definition-of-base-type-template

Syntax

```
%using  definition-of-base-type output-statement
```

Description

All references to the `output_substitution_sequence %type` in a `loop_over_parameters` are in the output as specified in *`output_statement`*. Default `output-statement` is `""` (empty).

Example

A	`%using A "unsigned char %name%index[%eLength]"`
AV	`%using AV "ERX_HVDATA %name%index"`
B	`%using B "unsigned char %name%index[%eLength]"`
BV	`%using BV "ERX_HVDATA %name%index"`
D	`%using D "unsigned char %name%index[ERX_GET_PACKED_LEN(7)]"`
F4	`%using F4 "float %name%index"`
F8	`%using F8 "double %name%index"`
I1	`%using I1 "signed char %name%index"`
I2	`%using I2 "short %name%index;"`
I4	`%using I4 "long %name%index;"`
K	`%using K "unsigned char %name%index[%eLength]"`
KV	`%usingKV "ERX_HVDATA %name%index"`
L	`%using L "unsigned char %name%index"`
N	`%using N "unsigned char %name%index[%before+%after]"`
NU	`%using NU "unsigned char %name%index[%before+%after]"`
P	`%using P "unsigned char %name%index[ERX_GET_PACKED_LEN(%before+%after)]"`
PU	`%using PU "unsigned char %name%index[ERX_GET_PACKED_LEN(%before+%after)]"`
T	`%using T "unsigned char %name%index[ERX_GET_PACKED_LEN(13)]"`

definition-of-base-type

Syntax

```
A | AV | B | BV | D | F4 | F8 | I1 | I2 | I4 | K | KV | L | N | NU | P | PU | T
```

Description

For a description of the definition-of-base-type see *IDL Data Types*.

A	Reference to IDL data type Alphanumeric.
AV	Reference to IDL data type Alphanumeric variable length
B	Reference to IDL data type Binary.
BV	Reference to IDL data type Binary variable length.
D	Reference to IDL data type Date.
F4	Reference to IDL data type Floating point (small).
F8	Reference to IDL data type Floating point (large).
I1	Reference to IDL data type Integer (small).
I2	Reference to IDL data type Integer (medium).
I4	Reference to IDL data type Integer (large).
K	Reference to IDL data type Kanji.
KV	Reference to IDL data type Kanji variable length.
L	Reference to IDL data type Logical.
N	Reference to IDL data type Unpacked decimal.
NU	Reference to IDL data type Unpacked decimal unsigned.
P	Reference to IDL data type Packed decimal.
PU	Reference to IDL data type Packed decimal unsigned.
T	Reference to IDL data type Time.

definition-of-direction-template

Syntax

```
%using  %direction output-statement output-statement output-statement
```

Description

All references to the `output_substitution_sequence %direction` in a `loop_over_parameters` are in the output as specified in `output-statement`. If a parameter is of direction `IN`, the first `output-statement` will be used, the second `output-statement` will be used for direction `OUT` and the third for `INOUT` (see ***`attribute-list`***). Default

`output-statement` is `"" "" ""` (empty).

Example

```
%using %direction "In" "Out" "In Out"
```

definition-of-group-template

Syntax

```
%using G output-statement output-statement
```

Description

If the parameter is a group (see *`group-parameter-definition`*) in a `loop_over_parameters`, all references to the `output_substitution_sequence %type` will be written into the output as specified in the `output-statements`. The first `output-statement` is the group prefix, typically the data type of the target programming language. The second `output-statement` is the group suffix which usually indicates the end of the group. Default `output-statement` is `""` (empty).

Example

```
%using G "struct {" "%outBlank} %name;"
```

definition-of-index-template

Syntax

```
%using  %index output-statement output-statement output-statement output-statement
```

Description

All references to the `output_substitution_sequence %index` in a `loop_over_parameters` will be written into the output as specified in the `output-statements`. According to the IDL (see *`array-definition`*), up to 3 dimensions are supported. The first `output-statement` is for scalar parameters, the second `output-statement` for 1-dimensional arrays, the third `output-statement` for 2-dimensional arrays and the fourth `output-statement` for 3-dimensional arrays. Default `output-statement` is `""` (empty).

Example

```
%using %index "" "[%1_index]" "[%2_index][%1_index]"
        "[%3_index][%2_index][%1_index]"
```

definition-of-line-number-format-template

Syntax

```
%using %NumberLine output-statement
```

Description

All references to the `output_substitution_sequence %LibCount`, `%ProgCount` and `%NameCount` are written into the output as specified in the `output-statements`. The `output-statement` uses the C `printf` format notation. Default `output-statement` is `%u`.

Example

```
%using %NumberLine "%.4u"
```

definition-of-member-separator-template

Syntax

```
%using %member output-statement
```

Description

Specify a template for a fully qualified name of parameters. All references to the `output_substitution_sequence %member` in a `loop_over_parameters` are written into the output as specified in the `output-statements`. The IDL Compiler builds an internal tree hierarchy of parameters, structures and groups. If a parameter has a parent, it will be inserted before the fully qualified name. If a parameter has no parent, the `output-statement` will not be used. Default `output-statement` is `""` (empty).

Example

```
%using %member "%name%Index."
```

definition-of-names-format-template

Syntax

```
%using  %Format output-statement
```

Description

Specify a template for library, program or parameter name strings. All references to the `output_substitution_sequence %OutputLevel` in a `loop_over_parameters` are written into the output as specified in the `output-statements`. The `output-statement` uses the C `printf` format notation. Default `output-statement` is `%s`.

Example

```
%using %Format "%s.ext"
```

definition-of-OutBlank-template

Syntax

```
%using %outBlank output-statement
```

Description

The `output-statement` definition replaces the blank (default), which will be used with the `%outBlank` statement. The statement will not be interpreted and will be used as it is. You cannot write an expression (`string_with_expression_contents`) or a variable (`variable_of_type_string string`) in this output statement. See the table entry *`definition-of-OutBlank-template`* in the section *`output_substitution_sequence`* for further information.

Example

```
%using %outBlank "\t"
```

definition-of-nest-level-format-template

Syntax

```
%using  %OutputLevel output-statement
```

Description

Specify a template for nesting level strings. The nesting level is the depth where a parameter is specified. All references to the `output_substitution_sequence %OutputLevel` in a `loop_over_parameters` are written into the output as specified in the `output-statements`. The `output-statement` uses the C `printf` format notation. Default `output-statement` is `%using`.

Example

```
%using %OutputLevel "%u"
```

definition-of-parent-identifier-template

Syntax

```
%using %Xparent output-statement output-statement
```

Description

Specify a template for the parent. All references to the `output_substitution_sequence %Xparent` in a `loop_over_parameters` are written into the output as specified in the `output-statements`. If a parameter has no parent and the second `output-statement` is not empty, the second `output-statement` is used, otherwise, the first. The first `output-statement` uses the C `printf` format notation. Default is `"%u" ""` (second statement is empty).

Example

```
%using %Xparent "%d" "ERX_NO_PARENT_V2"
```

definition-of-parent-index-template

Syntax

```
%using %Index output-statement output-statement output-statement output-statement
```

Description

Specify a template for a parameter's parent index. The IDL Compiler builds an internal tree hierarchy of parameters, structures and groups. A parameter's immediate parent in this hierarchy can be an array. All references to the `output_substitution_sequence %Index` in a `loop_over_parameters` are written into the output source code as specified in the `output-statements`. The `output-statement` describes the syntax for defining arrays of up to 3 dimensions as defined by the IDL (see *`array-definition`*). The `Output_substitution_sequence %Index` (uppercase I) is very similar to the `output_substitution_sequence %index` (defined with *`definition-of-index-template`*), but is useful only for building member names using `output_substitution_sequence %member`. Default `output-statement` is `""` (empty).

Example

```
%using %Index "" "[0]" "[0][0]" "[0][0][0]"

%using %member "%Index."
```

definition-of-structure-template

Syntax

```
%using S output-statement
```

Description

If the parameter is a structure (see *`structure-parameter-definition`* *(IDL)*) in a `a loop_over_parameters`, all references to the `output_substitution_sequence %type` will be written into the output as specified in the `output-statements`. `"INCLUDE AS GROUP"` specified as the `output-statement` will embed all parameters (see *`parameter-data-definition`*) of the structure as if they were a group.

Example

```
%using S "INCLUDE AS GROUP"
```

definition-of-UnboundedArray-template

Syntax

```
%using UnboundedArray output-statement
%using UnboundedArray ""
```

Description

If the parameter is an unbounded array (see *array-definition*) in a loop_over_parameters, all references to the output_substitution_sequence %type will be written into the output as specified in the output-statements statement, i.e. the settings of the *definition-of-base-type* are overwritten. The first form overwrites the settings of the definition-of-base-type. The second form switches back to the definition-of-base-type settings.

Example

```
%using UnboundedArray "ERX_HARRAY"
```

error_statement

Syntax

```
%error output_character_sequence
```

Description

Use the error_statement to exit your template with an error message. The execution of the template will be stopped at this statement and the error message will be given to the caller. The %error statement can be used in the main template and in subtemplates as well. The execution of the whole template compiling process will be stopped regardless of the type of template it is used in.

Example

```
%if "$(TARGET)" <> "COBOL" && "$(TARGET)" <> "BATCH"
{
   %error "TARGET not supported."
}
```

execute_statement

Syntax

```
%execute output-statement [(parameter_list)] [return (return_list)]
```

Description

Use the %execute statement to include another template file in the current template file like a subprogram. You can outsource often-used code to an external template file. The executed template file uses a NEW CLEAN environment context. Only the %library, %program, %x_struct, %name, %LibCount, %ProgCount and %NameCount are known in the executed template file. No other variables or IDL parameters (output_substitution_sequence) are defined in the executed template file. You *must* for example have a %name loop (loop_over_parameters) to have access to an IDL parameter (output_substitution_sequence). All changes in the executed files will be lost after calling the

include file and will have no effect in the calling template.

The %execute statement can be called with a list of parameters (parameter_list). The parameter_list needs to be set in brackets. This list of parameters will be copied into the variables in the executed file in the following order: ?A, ?B, ?C, ..., for example a list of parameters in the %execute statement ("AA" "BB" "CC" "DD"). These parameters will be available in the following variables: "?A" => "AA", "?B" => "BB", "?C" => "CC", "?D" => "DD".

The %execute statement can have a "return" statement to return a list of parameters (return_list), as well. This *return_list* has to be returned with the *return_statement*. The count of the expected and the returned parameter must be the same and the type of the parameters must match. For example, if the expected return parameter is a *variable_of_type_string*, the return parameter type must be a variable_of_type_string and if the expected return parameter is a *variable_of_type_integer*, the return parameter type must be a variable_of_type_integer.

Example 1

```
%execute "subprog.tpl" ("?A" "&i" "10")

%execute "subprog.tpl" ("?Z" "%OutputLevel" "subprog" "10") return ("?C" "&f")
```

Example 2

```
File main.tpl
%assign A "Test variable A"
%assign B "Test variable B"
%assign C "Length of A and B is"
%compute i "#A"  ; length of the variable A
%compute j "#B"  ; length of the variable B
%execute "calc.tpl" ("?C" "&i" "&j") return ("?Z")
"?Z\n" ; print the returned variable
; the output should be "Length of A and B is 30"
File calc.tpl
; the execute parameters are in the following variables
; ?A => the ?C of the main template = "Length of A and B is"
; ?B => the &i of the main template = "15"
; ?C => the &j of the main template = "15"
%compute a "?B + ?C"
%assign T "?A &a"
%return ("?T")
```

file_handling_statement

Syntax

```
%file [output-statement]
```

Description

Use the %file statement to direct the output to a specific file. Only one file can be open at a time. If no file is open, all output goes to STDOUT by default. If output-statement is left blank, the file currently open is closed. All open files are implicitly closed if either a new open file statement is encountered or the IDL Compiler terminates.

Example

```
%file "%library.MAK" ; open a file
....
%file "" ; close the file
```

if_statement

Syntax

```
%if compare_expression statement [if-elif-extension] [%else statement]
```

Description

If *compare_expression* is TRUE, then interpret statement. If compare_expression is FALSE and if there is an `if-elif-extension`, interpret the `if-elif-extension`.

If all compare_expressions in all `if-elif-extensions` are FALSE and if there is an else block, interpret the statement in the `else` block.

Example

```
%if "%type%index" ""
     "\n"
%elif "%index" ""
     "(%type)\n"
%else
     "(%type%index)\n"
```

if_elif_extension

Syntax

```
%elif compare_expression statement [if-elif-extension]
```

Description

If *compare_expression* is TRUE, then interpret the statement. If `compare_expression` is FALSE and if there is an `if-elif-extension`, interpret the if-elif-extension.

Example

See lexical entity *if_statement*.

logical_compare_operator

Syntax

&&	logical AND operator
||	logical OR operator

Description

The logical operators perform logical AND (&&) and logical OR (||) operations. The logical AND operator has a higher priority than the logical OR operator.

Example

See the lexical entity *compare_expression*.

loop_statement

Syntax

```
loop_over_libraries | loop_over_parameters | loop_over_programs | loop_over_structures | loop_of_while
```

Description

Loop sequences instruct the IDL Compiler to loop through all occurrences of libraries (see *library-definition*), programs (see *program-definition*), structures (see *structure-definition*) and parameters (see *parameter-data-definition*).

Example

See lexical entities:

- *loop_over_libraries*
- *loop_over_parameters*
- *loop_over_programs*
- *loop_over_structures*
- *loop_of_while*

loop_over_libraries

Syntax

```
%library statement
```

Description

This loop statement instructs the IDL Compiler to loop through all occurrences of libraries (see *library-definition*).

Example

```
%library
{
   ....
}
```

loop_over_parameters

Syntax

```
%name statement
```

Description

This loop statement instructs the IDL Compiler to loop through all occurrences of parameters (see *parameter-data-definition*). A loop over parameters must be placed in a *loop_over_programs* or *loop_over_structures*.

Example

```
%name
{
   ....
}
```

loop_over_programs

Syntax

```
%program statement
```

Description

This loop statement instructs the IDL Compiler to loop through all occurrences of programs (see *program-definition*). A loop over programs must be placed in a *loop_over_libraries*.

Example

```
%program
{
   ....
}
```

loop_over_structures

Syntax

```
%x_struct statement
```

Description

This loop statement instructs the IDL Compiler to loop through all occurrences of structures (see *structure-definition*). A loop over structures must be placed in a *loop_over_libraries* or *loop_over_programs*. In a *loop_over_libraries* all structures in the library (see *library-definition*) are accessed. In a loop_over_programs all structures in the current program (see *program-definition*) are accessed.

Example

```
%structure
{
   ....
}
```

loop_of_while

Syntax

```
%while compare_expression statement
```

Description

Loop while *`compare_strings`* is TRUE.

Example

```
%compute i "0"
%while "&i" < "10"
{
   ....
  %compute i "&i + 1"
}
```

message_statement

Syntax

```
%message output_character_sequence
```

Description

Use the `message_statement` to notify the template user with a message. The `output_character_sequence` will report as the message on the console.

Example

```
%if "%eLength" > "32766"
{
  %message "Maximum length for %type usually 32766"
}
```

output

Syntax

```
output_character_sequence | output_escape_sequence  | output_formatting_sequence|  output_of_variable |
output_substitution_sequence
```

Description

See lexical entities:

- *output_character_sequence*
- *output_escape_sequence*
- *output_formatting_sequence*
- *output_of_variable*
- *output_substitution_sequence*

Example

See lexical entities:

- *output_character_sequence*
- *output_escape_sequence*
- *output_formatting_sequence*
- *output_of_variable*
- *output_substitution_sequence*

output_character_sequence

Description

Simple sequences of characters not matching other output such as *output_escape_sequence*, *output_formatting_sequence*, *output_of_variable* or *output_substitution_sequence* form a character sequence.

Example

```
"This is a character sequences in an output statement."
```

output_control_ims

Syntax

```
%IMS+ | %IMS- | %IMS
```

Description

The `IMS` flag. If this flag is set, the parameter in the `loop_over_parameters`, that are marked with the `IMS` attribute in the IDL file will also be taken into consideration. If this flag is off, all parameters that are marked with the `IMS` attribute will be ignored. If `+` or `-` are not specified, the flag will be toggled. The default is off.

Example

```
%IMS+
%library
{
   %program
   {
      %name
      {
         "%name"
      }
   }
}
```

output_control_imsonly

Syntax

```
%IMSONLY+ | % IMSONLY - | % IMSONLY
```

Description

The `IMSONLY` flag. If this flag is set, parameters in the `loop_over_parameters` that are not marked with the `IMS` attribute in the IDL file will be ignored. If this flag is off, the `loop_over_parameters` will work as usual. If `+` or `-` are not specified, the flag will be toggled. The default is off.

Example

```
%IMSONLY+
%library
{
   %program
   {
      %name
      {
         "%name - this parameter has an ims attribute"
      }
   }
}
```

output_control_lower_upper

Syntax

(Defaults are underlined.)

```
%UpperCase+    | %UpperCase-    | %UpperCase
%UpperCasePgm+ | %UpperCasePgm- | %UpperCasePgm
%LowerCase+    | %LowerCase-    | %LowerCase
```

Description

`UpperCase`	Name uppercase flag. If this flag is set, `%name substitution_sequences` written to the output will be converted to uppercase. If no `+` or `-` is specified, the flag will be toggled. The default is off.
`UpperCasePgm`	Program uppercase flag. If this flag is set, `%program substitution_sequences` written to the output will be converted to uppercase. If no `+` or `-` is specified, the flag will be toggled. The default is off.
`LowerCase`	Name lowercase flag. If this flag is set, `%name substitution_sequences` written to the output will be converted to lowercase. If no `+` or `-` is specified, the flag will be toggled. The default is on.

Example

`UpperCase`	`%UpperCase+`	set name uppercase flag on
	`%UpperCase-`	set name uppercase flag off
	`%UpperCase`	toggle name uppercase flag
`UpperCasePgm`	`%UpperCasePgm+`	set program uppercase flag on
	`%UpperCasePgm-`	set program uppercase flag off
	`%UpperCasePgm`	toggle program uppercase flag
`LowerCase`	`%LowerCase+`	set name lowercase flag on
	`%LowerCase-`	set name lowercase flag off
	`%LowerCase`	toggle name lowercase flag

output_control_sanitize

Syntax

(Defaults are underlined.)

```
                    %Sanitize+ | %Sanitize- | %Sanitize %SanitizeCamelCased+ | %SanitizeCamelCased- |
%SanitizeCamelCased %SanitizeCobol+ | %SanitizeCobol- | %SanitizeCobol %SanitizePascalCased+ |
%SanitizePascalCased- | %SanitizePascalCased
```

Description

`Sanitize`	Sanitize flag for C programming language. If this flag is set, `%x_struct`, `%u_struct`, `%name`, `%program` and `%library` substitution sequences written to the output will be forced to follow C conventions. The special characters '#', '$', '&', '+', '-', '.', '/' and '@' in parameter names permitted in the IDL file will be converted to underscores '_' to produce valid C names. If + or - are not specified, the flag will be toggled. The default is on.
`SanitizeCamelCased`	Sanitize flag for C# programming language. If this flag is set, `%x_struct`, `%u_struct`, `%name`, `%program` and `%library` substitution sequences written to the output will be forced to follow camel cased naming conventions as they are used in C#. The special characters '#', '$', '&', '+', '-', '.', '/', '@' and '_' in parameter names permitted in the IDL file will be removed. The character following the special character will be converted to uppercase and all other characters to lowercase. The very first character within `%name`, `%program`, and `%library` substitution sequences will be converted to lowercase. `%UpperCase-` and `%LowerCase-` must be set also to have `CamelCased` names. If + or - are not specified, the flag will be toggled. The default is off.
`SanitizeCobol`	Sanitize flag for COBOL programming language. If this flag is set, `%x_struct`, `%u_struct`, `%name`, `%program` and `%library` substitution sequences written to the output will be forced to follow COBOL conventions. The special characters '#', '$', '&', '+', '.', '/', '@' and '_' in parameter names permitted in the IDL file will be converted to hyphen '-' to produce valid COBOL names. . If a parameter name starts with a digit, e.g. '1', it is prefixed with the character 'P'. If + or - are not specified, the flag will be toggled. The default is off.
`SanitizeDCOMWrapper`	Sanitize flag for DCOM Wrapper. If this flag is set, `%x_struct`, `%u_struct`, `%name`, `%program` and `%library` substitution sequences written to the output will be forced to follow DCOM conventions. The special characters '#', '$', '&', '+', '-', '.', '/' and '@' in parameter names permitted in the IDL file will be converted to underscores '_' to produce valid DCOM names. All preceding underscores in parameter names are deleted. If a parameter name starts with a digit, e.g. '1', it is prefixed with the character 'P'. If + or - are not specified, the flag will be toggled. The default is on.

SanitizePascalCased	Sanitize flag for C# programming language. If this flag is set, %x_struct, %u_struct, %name, %program and %library substitution sequences written to the output will be forced to follow Pascal-cased naming conventions as they are used in C#. The special characters '#', '$', '&', '+', '-', '.', '/', '@' and '_' in parameter names permitted in the IDL file will be removed. The character following the special character will be converted to uppercase and all other characters to lowercase. The very first character in the %name, %program and %library substitution sequences will be converted to uppercase. %UpperCase- and %LowerCase- must be set also to have PascalCased names. If + or - are not specified, the flag will be toggled. The default is off.
SanitizePLI	Sanitize flag for the PL/I programming language. If this flag is set, %x_struct, %u_struct, %name, %program and %library substitution sequences written to the output will be forced to follow PL/I conventions. The special character '&', '+', '-', '.' and '/' in parameter names permitted in the IDL file will be converted to underscores '_' to produce valid PL/I names. If + or - are not specified, the flag will be toggled. The default is off.

Example

Sanitize	%Sanitize+	set Sanitize flag for C variable names on
	%Sanitize-	set Sanitize flag for C variable names off
	%Sanitize	toggle Sanitize flag for C variable names
SanitizeCamelCased	%SanitizeCamelCased+	set Sanitize flag for C# parameter names on
	%SanitizeCamelCased-	set Sanitize flag for C# parameter names off
	%SanitizeCamelCased	toggle Sanitize flag for C# parameter names
SanitizeCobol	%SanitizeCobol+	set Sanitize flag for COBOL variable names on
	%SanitizeCobol-	set Sanitize flag for COBOL variable names off
	%SanitizeCobol	toggle Sanitize flag for COBOL variable names

SanitizeDCOMWrapper	%SanitizeDCOMWrapper+	set Sanitize flag for DCOM names on
	%SanitizeDCOMWrapper-	set Sanitize flag for DCOM names off
	%SanitizeDCOMWrapper	toggle Sanitize flag for DCOM names
SanitizePascalCased	%SanitizePascalCased+	set Sanitize flag for C# member names on
	%SanitizePascalCased-	set Sanitize flag for C# member names off
	%SanitizePascalCased	toggle Sanitize flag for C# member names
SanitizePLI	%SanitizePLI+	set Sanitize flag for PL/I member names on
	%SanitizePLI-	set Sanitize flag for PL/I member names off
	%SanitizePLI	toggle Sanitize flag for PL/I member names

output_control_statement

Syntax

```
output_control_lower_upper | output_control_sanitize |
        output_control_verbose
```

Description

Use the flags to force upper/lowercase conversions, C, C# or COBOL language conventions or, for example, for comments to be written into the output. The default setting when you do not code any `output_control_statements` are forced to follow the C programming language convention.

Example

See under the lexical entities:

- *`output_control_lower_upper`*

- *output_control_sanitize*
- *output_control_verbose*

output_control_verbose

Syntax

```
%verbose+ | %verbose- | %verbose
```

Description

Verbose flag. If this flag is set, template file `output-statements` will be written to the output without being interpreted, e.g. the `substitution_sequences` are output as is and not replaced by their meaning. If + or - are not specified, the flag will be toggled. The default is off.

Example

```
%verbose+
"/* This is file %program.c */"
"/* Please do not modify this file */"
%verbose
```

output_escape_sequence

Syntax

```
\\
```

Description

The escape character is used to change the meaning of the special characters &, ? and # back to their normal meaning. Special characters access variables using the *output_of_variable* lexical entity. With escape characters it is possible to insert a plain & by typing : `\\&`.

Example

```
"This string contains an ampersand \\&."
```

output_formatting_sequence

Syntax

Sequence	Meaning
`\n`	Newline
`\r`	Carriage return
`\t`	Horizontal tab
`\ddd`	ASCII character in octal notation, e.g. `\012` for the new line
`\xdd`	ASCII character in hex notation, e.g. `\x09` for the horizontal tab

Description

Formatting sequences are output control characters such as newline, backspace, etc. For characters in hexadecimal notation, the IDL Compiler ignores all leading zeros. It establishes the end of the hex-specified escape character when it encounters either the first non-hex character or more than two hex characters not including leading zeros. In the latter case, it reports an error and ignores all characters beyond the second one.

Example

```
"This string ends on this line.\n"
```

output_of_variable

Syntax

```
??variable_of_type_indexed_string | ## variable_of_type_indexed_string
        | &variable_of_type_integer | ?variable_of_type_string | #variable_of_type_string | $(...)
```

Description

This form of accessing variables can be used in `output_statements`. If it is used, the variable content is written to the output.

Example

`&a`	substitutes the integer variable with its number.
`?A`	substitutes the string variable with its contents.
`??A[0]`	substitutes the indexed string variable with its contents.
`#A`	substitutes the string variable with an integer of the length of its contents.
`##B[%OutputLevel]`	substitutes the indexed string variable with an integer of the length of its contents.
`$(TEMP)`	substitutes the option variable with its contents.

output_statement

Syntax

```
'"' { output } '"'
```

Description

output is a string consisting of an *output_character_sequence*, *output_escape_sequence*, *output_formatting_sequence*, *output_of_variable* or an *output_substitution_sequence* in any order.

Example

```
"This is simple output."
```

output_substitution_sequence

Syntax

Sequence	Meaning
%after	Inserts the digits after the decimal point of the current parameter (see *simple-parameter-definition*) into the output. This substitution sequence can only be used in an active *loop_over_parameters*, and if the current parameter is of data type N, NU, P, PU. Using it with other data types will lead to an error
%Alias	If a library alias name is used, inserts the library alias name of the current library (see *library-definition*) into the output. If no alias is provided, the library name (contents of %library) is provided in the %Alias substitution sequence. This substitution sequence can only be used in an active *loop_over_libraries*.
%before	Inserts the digits before decimal point of the current parameter (see *simple-parameter-definition*) into the output. This substitution sequence can be used only in an active *loop_over_parameters*, and if the current parameter is of data type N, NU, P, PU. Using it with other data types will lead to an error.
%Count	Inserts the output's current line number. The format is controlled by the *definition-of-names-format-template*.
%direction	Inserts the direction of the current parameter (see *parameter-data-definition*) into the output as specified with *definition-of-direction-template*. This substitution sequence can only be used in an active *loop_over_parameters*.

<table>
<tr><th>Sequence</th><th>Meaning</th></tr>
<tr><td>%eLength</td><td>
<table>
<tr><th>Data Type</th><th>Description</th></tr>
<tr><td>A,K,B,I,F,U</td><td>Logical length as set in the IDL file.</td></tr>
<tr><td>AV,KV,BV,UV</td><td>Maximum logical length as set in the IDL file.</td></tr>
<tr><td>L</td><td>Type L has no explicit logical length in the IDL file. However, the length is always set to 1.</td></tr>
<tr><td>N,NU,P,PU</td><td>Digits before and after decimal point in encoded form as given in the IDL file. Use the macro ERX_GET_DIGITS (see erx.h) to get the digits before the decimal point. Use the macro ERX_GET_DECIMALS (see erx.h) to get the number of digits after the decimal point.</td></tr>
<tr><td>T</td><td>Type T has no explicit logical length in the IDL file. However, the length is always set to 12.</td></tr>
<tr><td>D</td><td>Type D has no explicit logical length in the IDL file. However, the length is always set to 6.</td></tr>
</table>
</td></tr>
<tr><td>%file</td><td>Inserts the current filename into the output.</td></tr>
<tr><td>%Format</td><td>Inserts into the output the base name as given by the -F parameter on start of the The Software AG IDL Compiler. If no -F parameter is provided during start of the IDL Compiler, the base name of the idl-file without path and extension is inserted into the output.</td></tr>
<tr><td>%index</td><td>Inserts the index of the current parameter (see parameter-data-definition) into the output as specified with definition-of-parent-index-template. This substitution sequence can only be used in an active loop_over_parameters.</td></tr>
<tr><td>%LibCount</td><td>Inserts the total number of libraries (see library-definition) as given in the IDL file into the output. The format is controlled by the definition-of-line-number-format-template.</td></tr>
<tr><td>%library</td><td>Inserts the current library name (see library-definition) into the output. The inserted library name is controlled by output_control_statement. This substitution sequence can only be used in an active loop_over_libraries.</td></tr>
<tr><td>%member</td><td>Inserts the fully qualified member name: (parent...parent...child) into the output. The member name is controlled by definition-of-member-separator-template. This substitution sequence can only be used in an active loop_over_parameters.</td></tr>
</table>

Sequence	Meaning
`%Method`	If a program alias is used, inserts the program alias name of the current program (see *`program-definition`*) into the output. If no alias is provided, the program name (contents of `%program`) is provided. This substitution sequence can only be used in an active *`loop_over_programs`*.
`%name`	Inserts the current parameter name (see *`parameter-data-definition`*) into the output. The parameter name is controlled by *`output_control_statement`*. This substitution sequence can only be used in an active *`loop_over_parameters`*.
`%NameCount`	Inserts the total number of parameters (see *`parameter-data-definition`*) of the current program (see *`program-definition`*) into the output. The format is controlled by the `definition-of-line-number-format-template`. This substitution sequence can only be used in an active *`loop_over_programs`*.
`%outBlank`	Writes *n* blanks (*n* = nesting level of parameter) into the output according to the level of the current parameter (see *`parameter-data-definition`*). This substitution sequence can only be used in an active *`loop_over_parameters`*. You can replace the blank with another output statement by using (`%using %outBlank output-statement`).
`%OutputLevel`	Inserts the level of the parameter (see *`parameter-data-definition`*) into the output. This substitution sequence can only be used in an active *`loop_over_parameters`*.
`%ProgCount`	Inserts the total number of programs (see *`program-definition`*) in the current library into the output. The format is controlled by the `definition-of-line-number-format-template`. This substitution sequence can only be used in an active *`loop_over_libraries`*.
`%program`	Inserts the current program (see *`program-definition`*) name into the output. The output is controlled by *`output_control_statement`*. This substitution sequence can only be used in an active *`loop_over_programs`*.
`%type`	Inserts the parameter type (see *`parameter-data-definition`*) into the output. The parameter type is controlled by *`definition-of-base-type-template`*, `definition-of-group-template`, `definition-of-UnboundedArray-template` and `definition-of-structure-template`. This substitution sequence can only be used in an active *`loop_over_parameters`*.
`%size` (deprecated, should no longer be used!)	The `%size` substitution sequence is deprecated and should no longer be used. Use `%eLength` substitution sequence instead. Using `%size` will lead to an error.

Sequence	Meaning
`%TypeAttributes`	The substitution sequence `%TypeAttributes` produces a 2-byte bitmask indicating • for arrays (see *`simple-parameter-definition`*) whether dimensions are fixed or unbounded (see *`array-definition`*), • the `aligned` attribute (see *`attribute-list`*) for all types of parameters. The bitmask displays the following: • If an array and the 1st dimension is unbounded then bit 0 is `ON` rightmost. • If an array and the 2nd dimension is unbounded then bit 1 is `ON`. • If an array and the 3rd dimension is unbounded then bit 2 is `ON`. • If the `aligned` attribute is set then bit 3 is `ON`. For users of the EntireX RPC C Runtime the bitmask corresponds directly to the `ERXeAttributes` defined for the `ERX_PARAMETER_DEFINITION_V3`. (see *erx.h*) This substitution sequence can only be used in an active *`loop_over_parameters`*.
`%u_struct`	Inserts the name of the referenced structure (see *`structure-definition`*) into the output. The inserted name is controlled by *`output_control_statement`*. This substitution sequence can only be used in an active *`loop_over_parameters`* and only when the current parameter uses a structure as its type definition. See *`structure-parameter-definition`* *(IDL)*.
`%x_struct`	Inserts the name of the structure (see *`structure-definition`*) into the output. The inserted name is controlled by *`output_control_statement`*. This substitution sequence can only be used in an active *`loop_over_parameters`*.
`%Xparent`	Inserts the parameter's parent into the output. This substitution sequence can only be used in an active *`loop_over_parameters`*.
`%0_index`	Inserts the number of indices of the current parameter (see *`parameter-data-definition`*) into the output. This substitution sequence can only be used in an active *`loop_over_parameters`*.
`%1_index`	Inserts the count of elements in dimension 1 of the current parameter (see *`parameter-data-definition`*) into the output. The substitution sequence should be used for 1-dimensional parameters only (this can be checked with the `%0_index` substitution sequence). This substitution sequence can only be used in an active *`loop_over_parameters`*.

Sequence	Meaning
`%2_index`	Inserts the count of elements in dimension 2 of the current parameter (see *`parameter-data-definition`*) into the output. The substitution sequence should be used for 2-dimensional parameters only (this can be checked with the `%0_index` substitution sequence). This substitution sequence can only be used in an active *`loop_over_parameters`*.
`%3_index`	Inserts the count of elements in dimension 3 of the current parameter (see *`parameter-data-definition`*) into the output. The substitution sequence should be used for 3-dimensional parameters only (this can be checked with the `%0_index` substitution sequence). This substitution sequence can only be used in an active *`loop_over_parameters`*.
`%SameLineComment`	Inserts the text of the comment line of the current parameter from the IDL file. Use the parameter properties of the IDL Editor to set this comment line in the IDL file.
`%SVMMetaData`	Inserts the metadata part contained in a related server mapping file (see *Server Mapping Files for Natural*) of the current IDL program into the output. This substitution sequence can only be used in an active *`loop_over_programs`*. If there is no related mapping file, an empty string is inserted
`%SVMFormatArea`	Inserts the format area contained in a related server mapping file of the current IDL program into the output. This substitution sequence can only be used in an active *`loop_over_programs`*. If there is no related mapping file, an empty string is inserted
`%SVMValueArea`	Inserts the value area contained in a related server mapping file of the current IDL program into the output. This substitution sequence can only be used in an active *`loop_over_programs`*. If there is no related server mapping file, an empty string is inserted
`%SVMStringArea`	Inserts the string area contained in a related server mapping file of the current IDL program into the output. This substitution sequence can only be used in an active *`loop_over_programs`*. If there is no related server mapping file, an empty string is inserted
`%SVMRpcProtocol`	Inserts the RPC protocol version contained in a related server mapping file of the current IDL program into the output. This substitution sequence can only be used in an active *`loop_over_programs`*. If there is no related server mapping file, an empty string is inserted

Description

Substitution sequences are substituted by their actual contents during generation.

Example

```
"This is a substitution sequence containing the library: %library."
```

parameter_list

Syntax

```
(parameter_list)
```

Description

The `parameter_list` is an unnumbered count of parameters. This list of parameters needs to be set in brackets. Each parameter needs to be set in quotation marks and will be interpreted before using. Parameters will be separated by blanks. The `parameter_list` can be defined as an empty list, in which case the `return_lis`t needs to be defined only with the brackets "`()`".

Example

```
("param" "10" "?A" "&i" "%0_index" "%OutputLevel")
```

return_list

Syntax

```
(return_list)
```

Description

The `return_list` is an unnumbered count of parameters. This list of return parameters needs to be set in brackets. Each return parameter needs to be set in quotation marks. Parameters will be separated by blanks. Only `variable_of_type_string` and `variable_of_type_integer` are allowed in this list. `variable_of_type_indexed_string` is not allowed. It is also not allowed to use any constant string. The `return_list` can be defined as an empty list, in which case the `return_list` needs to be defined only with the brackets "`()`".

Example

```
("?A" "?Z" "&i" "&n")
```

return_statement

Syntax

```
%return (parameter_list)
```

Description

The `return_statement` will be used to return "return parameters" from an executed subtemplate. No statements after the `return_statement` will be executed. The subtemplate will return to the main template with this statement. If the `return_statemen`t has been placed in the main template, the execution of the template will be stopped as normal at this point.

Example

```
%return ("param" "10" "?A" "&i" "%0_index" "%OutputLevel")
```

statement

Syntax

```
block | control_statement | output_statement
```

Description

These are the 3 basic types of statements used in a template. A block is a sequence of statements. `Output_statements` create the output. `Control_statements` determine the processing logic.

Example

See under the lexical entities *`control_statement`* and *`output_statement`*.

string

Syntax

```
'"' { output } '"'
```

Description

Any kind of output can be used to form a string. A string is used to form the condition criteria in a *`compare_strings`* lexical entity used e.g. in *`if_statement`* and *`loop_of_while`*. A string is not written to the output.

Example

```
"String with contents of variable $A"
```

string_with_expression_contents

Syntax

```
'"' { output } '"'
```

Description

Any kind of output can be used to form `String_with_expression_contents`. However, this kind of string must adhere to the rules of an expression. A `string_with_expression_rules` is not written to the output.

Supported mathematical operations are:
+ addition
- subtraction
* multiplication

/ division
mod modulo operation; computes the remainder after dividing its first operand by its second

Supported bit operations are:
`and` bitwise `AND` operation
`or` bitwise `OR` operation
`xor` bitwise `XOR` operation

Precedence of operators:
*, /, %
+, -
`xor`, `and`, `or`

You may control the precedence of the operation with brackets.

Example

```
%compute a "%OutputLevel + 1"
%compute b "%OutputLevel * 10"
%compute c "%TypeAttribute mod 3"
%compute d "(%TypeAttribute and 7) * 10"
%compute e "%TypeAttribute or 1"
%compute f "%TypeAttribute xor 3"
```

substring_statement

Syntax

```
%substring  variable_of_type_string string from_position length |
%substring  variable_of_type_indexed_string string from_position length |
```

Description

Extract from the source variable *`string`* the substring `from_position` up to the length to *`variable_of_type_string`* or *`variable_of_type_indexed_string`*.

The parameters `from_position` and `length` are of *`variable_of_type_integer`*. It is *not* possible to use a *`string_with_expression_contents`* for `from_position` and `length`. For `length`, the constant `all` or `ALL` can be used to extract the rest of the source string starting from `from_position`. The first position of the source string is 0.

If `length` is longer than the length of the substring to extract, the available substring from from_position to the end of the string will be assigned (same as using the constant "all" for length). If the from_position is been higher as the length of the string, an empty string will be assigned. If the value of from_position or length is lower than 0, an error will occur.

Example

```
%compute f "0"
%compute l "32 + 10"
%substring A "These are all characters before position 42 and these are all characters after position 42" "&f" "&l"
%compute f "&f + &l"
%compute l "100"
%substring A[1] "These are all characters before position 42 and these are all characters after position 42" "&f" "&l"
```

After execution, the variable A contains the string "These are all characters before position 42" and variable A[1] contains the string " and this are all characters after position 42".

UnsupportedProgram_statement

Syntax

```
%UnsupportedProgram output_character_sequence
```

Description

Use the `UnsupportedProgram_statement` to notify the IDL Compiler that the current program in the current library is not supported and needs to be ignored in further processing. The template writer can inform the template user of the reason why this program is not supported with the `output_character_sequence`. Usually the template writer will mark a program as unsupported if the program contains unsupported data type in the IDL definition for the target programming language. The `output_character_sequence` will report as a message on the console.

This statement must be embedded in `loop_over_libraries` and `loop_over_programs`. Furthermore it cannot be used if a file was already open using `file_handling_statement`.

Example

```
%using K  "K"
%library
{
   %program
   {
      %name
      {
        %if "%type" = "K"
         {
           %compute z "%eLength /2"
           %compute z "&z *2"
           %if "&z" <> "%eLength"
           {
             %UnsupportedProgram "Length for %type fields must be even."
           }
         }
      }
   }
}
```

variable_index

Syntax

```
string_with_expression_contents
```

Description

Any `variable_index` follows the rules of a `string_with_expression_rules`. Additionally, the result of the expression is restricted to the range 0 - 8, i.e. indices must be in the range of 0 - 8. The `variable_index` is not written to the output.

Example

```
"2"
"%OutputLevel"
"&A + 1"
```

variable_name

Syntax

The variable name can be A - Z and a - z.

Description

Variable names are not case-sensitive. `variable_of_type_integer`, `variable_of_type_string` and `variable_of_type_indexed_string` are distinct variables.

Example

```
A
B
a
b
```

variable_of_type_indexed_string

Syntax

```
variable-name[variable_index]
```

Description

`variable_of_type_indexed_string` are always correctly initialized to blanks.

Example

```
A[0]
B[%OutputLevel]
```

variable_of_type_integer

Syntax

```
variable-name
```

Description

`variable_of_type_integer` are initialized to zero.

Example

```
A
B
a
b
```

variable_of_type_string

Syntax

```
variable-name
```

Description

`variable_of_type_string` are always initialized to blanks.

Example

```
A
B
a
b
```