webMethods EntireX

Software AG IDL Editor

Version 9.6

April 2014

This document applies to webMethods EntireX Version 9.6.

Specifications contained herein are subject to change and these changes will be reported in subsequent release notes or new editions.

Document ID: EXX-EEXXIDLEDITOR-96-20140628

Table of Contents

Software AG IDL Editor

Software AG IDL (Interface Definition Language) is a language that lets a program or object (the client) written in one language communicate with another program written in another language (the server). An interface definition language works by requiring a program's interfaces to be described in an interface object or slight extension of the program that is compiled into it.

Software AG IDL is used to define an interface between the client and the server. These definitions are contained in a Software AG IDL file. IDL compilers read IDL files and generate interface objects or descriptions from the definitions they contain.

The Software AG IDL Editor is a syntax-aware editor for Software AG IDL files. A content outline view enables operations on the IDL tree. Other features include syntax highlighting, content assist and a Problems view for resolving IDL syntax errors.

Introduction	This document gives an introduction and overview of features of the IDL Editor.
Using the IDL Editor	This document describes the usage of the IDL Editor; starting the editor; views and context menu.
IDL File	This document contains a descriptive introduction to IDL data files; IDL data types; rules for coding IDL files.
IDL Grammar	This document explains the syntax of IDL files in a formal notation.
IDL Compiler	Starting the IDL Compiler; usage examples; writing your own wrappers and stubs.
Template Files	Writing your own template files for Software AG IDL Compiler.
Grammar for IDL Template Files	Grammar for IDL template files.

1 Introduction to the Software AG IDL Editor

The Software AG IDL Editor is a syntax-aware editor for Software AG IDL files. A content outline view enables operations on the IDL tree. Other features include syntax highlighting, content assist and a Problems view for resolving IDL syntax errors.

Introduction

The IDL Editor is used for processing IDL files (files with extension ".idl").

To start the IDL Editor

- Select an existing IDL file and open this using the context menu, or double-click on the file.

  Or:

  Start the IDL Editor as any other Eclipse new wizard.

In addition to a Text view, the editor supports the following views:

Outline View
In the Outline view, the structure of the IDL file is displayed in a tree structure. The context menu of the Outline view provides a variety of commands for manipulating the structure of the IDL. See *IDL Editor Outline View*.

Properties View
The Properties view displays various attributes of the element selected in the Text view or Outline view. You can also edit these attributes in this view. See *IDL Editor Properties View*.

Problems View
The Problems view describes syntax errors in the IDL file. See *IDL Editor Problems View*.

Features of the IDL Editor

Content Assist

Enables text-dependent completion when requested. Content assist offers a pop-up from which you can choose one of the suggestions.

Syntax Highlighting

Certain parts of the IDL document are displayed with a color coding:

- purple: IDL keywords
- brown: comments
- green: string constants.

2 Using the Software AG IDL Editor

The Software AG IDL Editor is a syntax-aware editor for Software AG IDL files. A content outline view enables operations on the IDL tree. Other features include syntax highlighting, content assist and a Problems view for resolving IDL syntax errors.

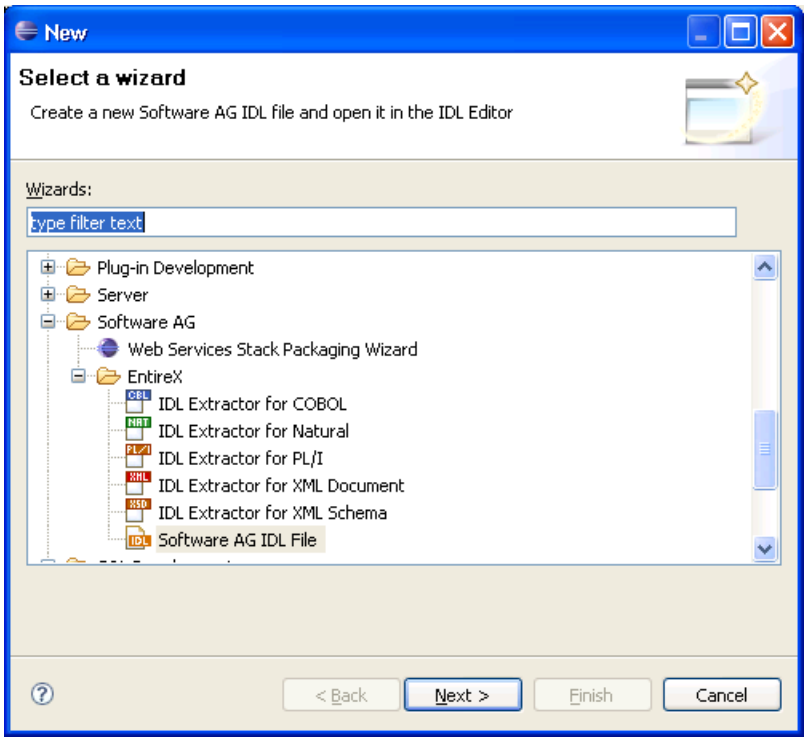
Starting the IDL Editor

To start the IDL Editor

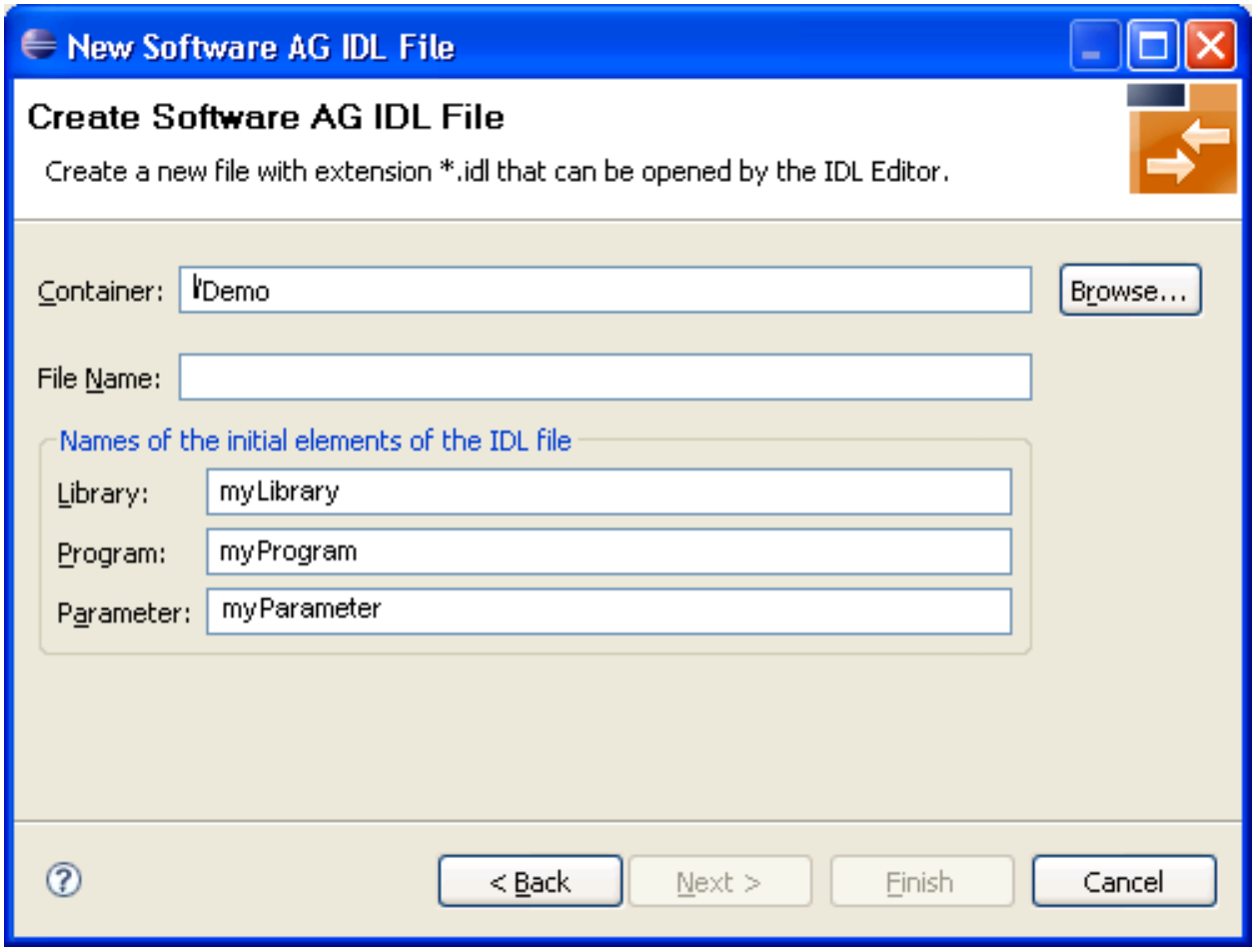
- Select an existing IDL file and open this using the context menu, or double-click on the file.

  Or:

  Start the IDL Editor as any other Eclipse new wizard.

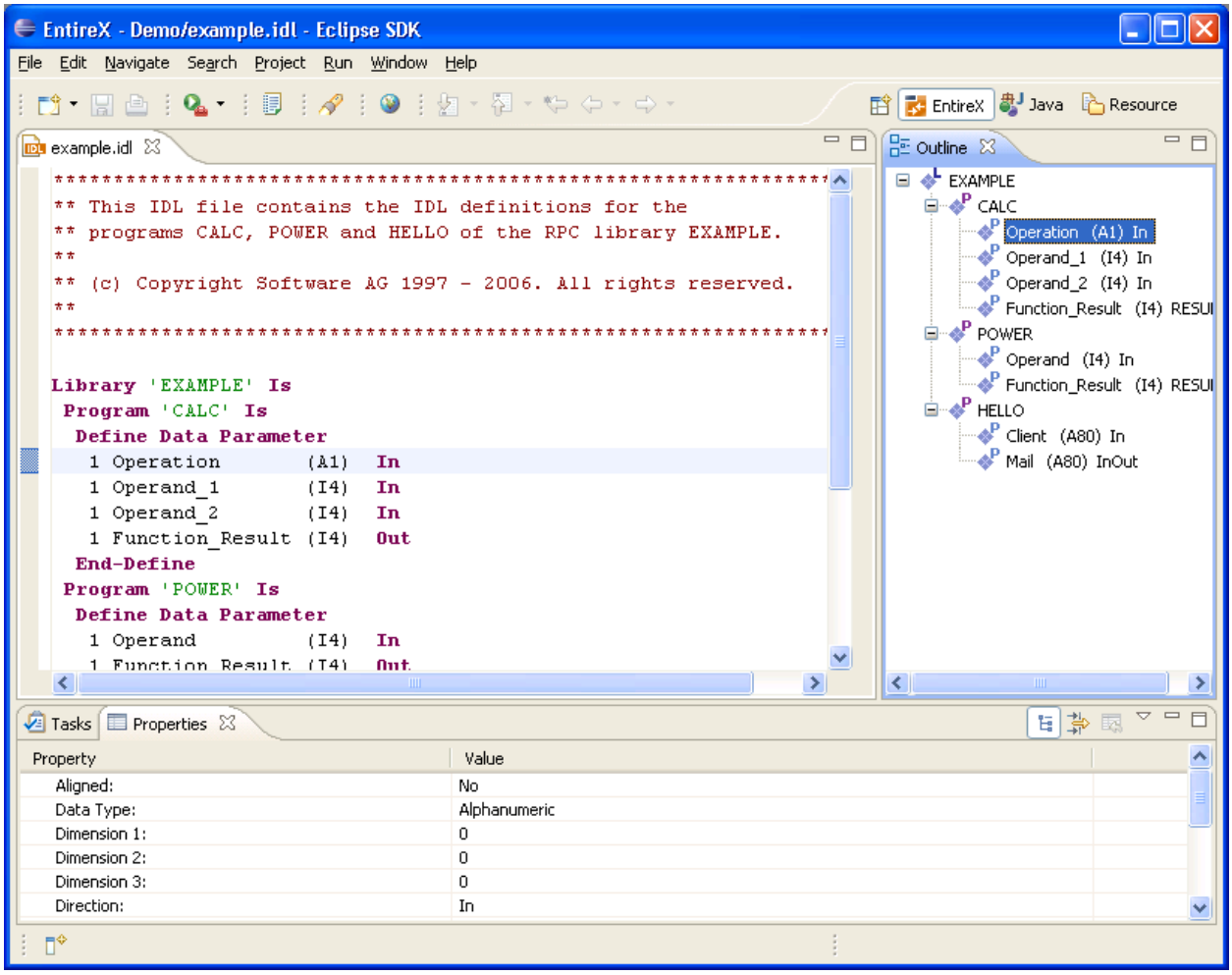


Specify the name and the container of the new IDL file you want to create. The container is a project or a folder in a project. If the container does not exist, it will be created interactively. Enter the names of a first library, a first program and a first parameter to get an initial IDL file. The names and the properties can be changed later. "myLibrary", "myProgram" and "myParameter" are provided as default values.



The IDL file is then displayed in the IDL Editor. In addition to a Text view, the editor supports the following views:

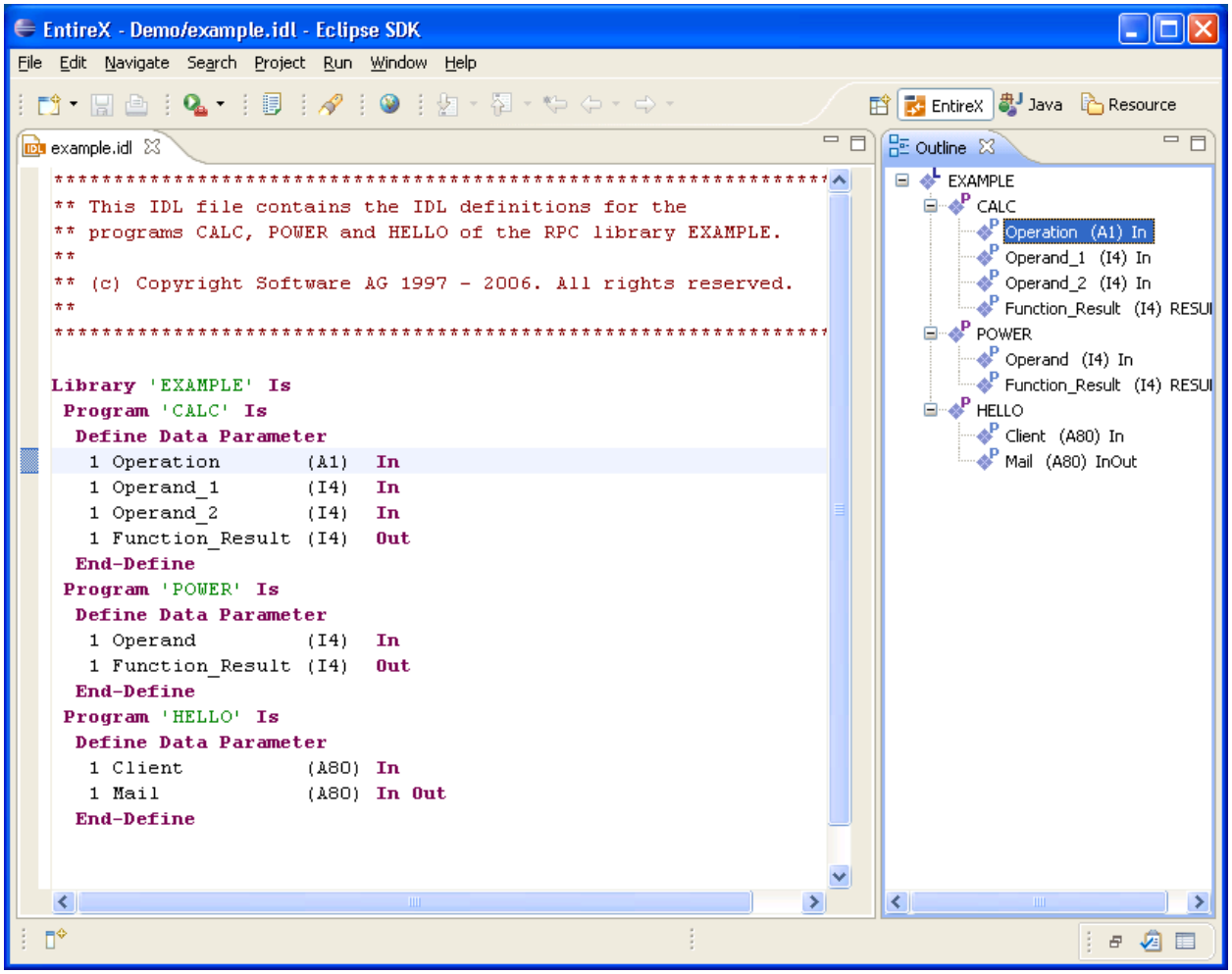
- *IDL Editor Outline View*
- *IDL Editor Properties View*
- *IDL Editor Problems View*


EntireX - Demo/example.idl - Eclipse SDK
File Edit Navigate Search Project Run Window Help
EntireX Java Resource
example.idl
** This IDL file contains the IDL definitions for the
** programs CALC, POWER and HELLO of the RPC library EXAMPLE.
** (c) Copyright Software AG 1997 - 2006. All rights reserved.
Library 'EXAMPLE' Is
Program 'CALC' Is
Define Data Parameter
1 Operation (A1) In
1 Operand_1 (I4) In
1 Operand_2 (I4) In
1 Function_Result (I4) Out
End-Define
Program 'POWER' Is
Define Data Parameter
1 Operand (I4) In
Outline
EXAMPLE
CALC
Operation (A1) In
Operand_1 (I4) In
Operand_2 (I4) In
Function_Result (I4) RESUI
POWER
Operand (I4) In
Function_Result (I4) RESUI
HELLO
Client (A80) In
Mail (A80) InOut
Tasks
Properties
Property
Value
Aligned: No
Data Type: Alphanumeric
Dimension 1: 0
Dimension 2: 0
Dimension 3: 0
Direction: In

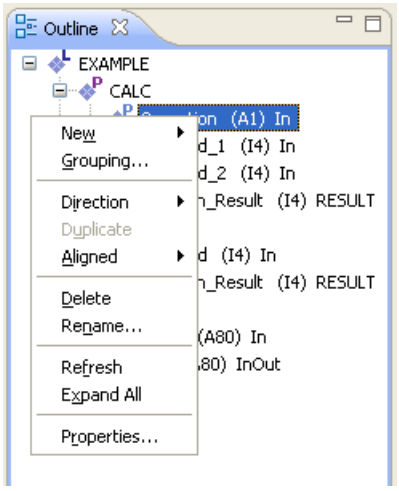
IDL Editor Views

IDL Editor Outline View

In the Outline view, the structure of the IDL file is displayed in a tree structure. The nodes are the elements of the IDL: libraries, programs, structures and parameters. The context menu of the Outline view provides a variety of commands for manipulating the structure of the IDL.



Context Menu of the Outline View



Command	Description
New	■ **Library** Inserts a library before the selected library. Enabled only for libraries. ■ **Program** Inserts a program at the first position of the current library if a library is selected. If a progam is selected, a new program is inserted before the selected program. The same applies if a structure is selected. If a parameter is selected, a new program is inserted before the program or structure containing the selected parameter. ■ **Structure** Inserts a structure at the first position of the current library if a library is selected. If a program is selected, the new structure is inserted before the selected program. The same applies if a structure is selected. If a parameter is selected, a new structure is inserted before the program or structure containing the selected parameter. ■ **Parameter** Inserts a new parameter at the first position of the current program or structure if a program or structure is selected. If a parameter is selected, the new parameter is inserted before the selected parameter. The `New` command applies only to single selections.
Grouping...	Inserts a group that contains the selected parameters.
Direction	■ **InOut** Changes the direction to InOut. ■ **In** Changes the direction to In.

Command	Description
	■ **Out** Changes the direction to Out. If a program is selected, the direction of all parameters in this program is changed.
Duplicate	Duplicates the selected program and creates a unique name for this program.
Aligned	■ **On** Parameter is aligned. ■ **Off** Parameter is not aligned.
Delete	Deletes the node.
Rename	Renames the node.
Refresh	Refreshes the whole tree, that is, makes it synchronized with the text in the editor.
Expand All	Opens the nodes of the tree.
Properties...	Displays the properties of the selected node.

Outline View Drag-and-Drop

Drag-and-drop operation is enabled for the tree structure of the IDL file (represented in the Outline view). That is, you can drag an individual library, structure, program or parameter to a target element. You cannot drag more than one object at a time.

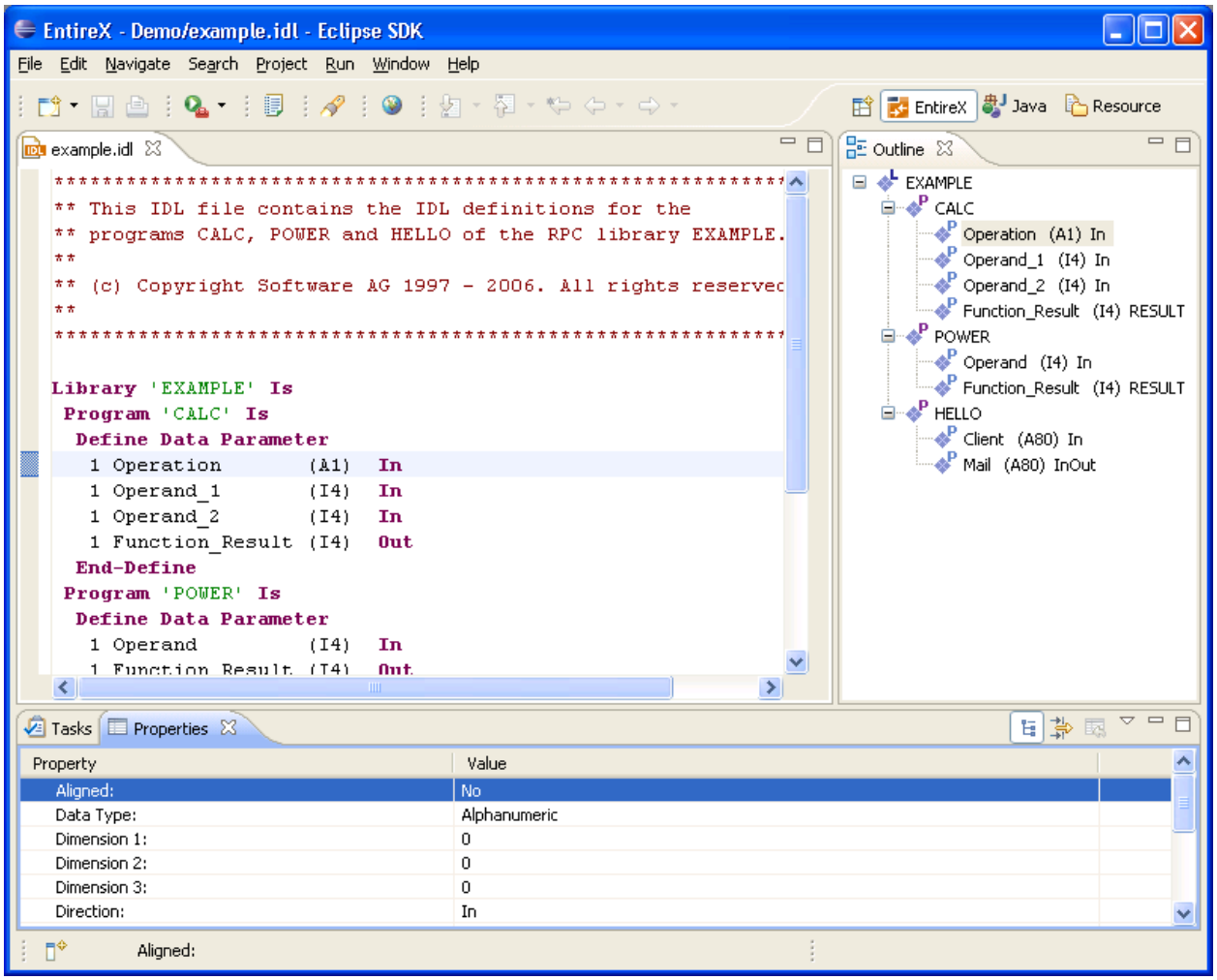
The table below describes the permitted drag-and-drop functions, where **Drag Source** is the element being dragged and the **Drop Target** is the element that receives the drag source.

Drop Target / Drag Source	Parameter, Group	Program	Structure	Library
Parameter, Group	x	x	x	
Program		x	x	x
Structure		x	x	x
Library				x

For example, a parameter can be dragged to a structure, but not to a library.

IDL Editor Properties View

The Properties view displays various attributes of the element selected in the Text view or Outline view. You can also edit these attributes in this view.



Library and Program

Property	Description
Alias	Alias of the library or program.
Name	Name of the library or program.
Preceding comment lines	Any comment in a preceding line or lines that applies to this line is displayed here. You can also add here a comment to this library or program in a definition line.
Same-line comment	Any comment in the current line is displayed here. You can also add a comment to the current line.

Structure

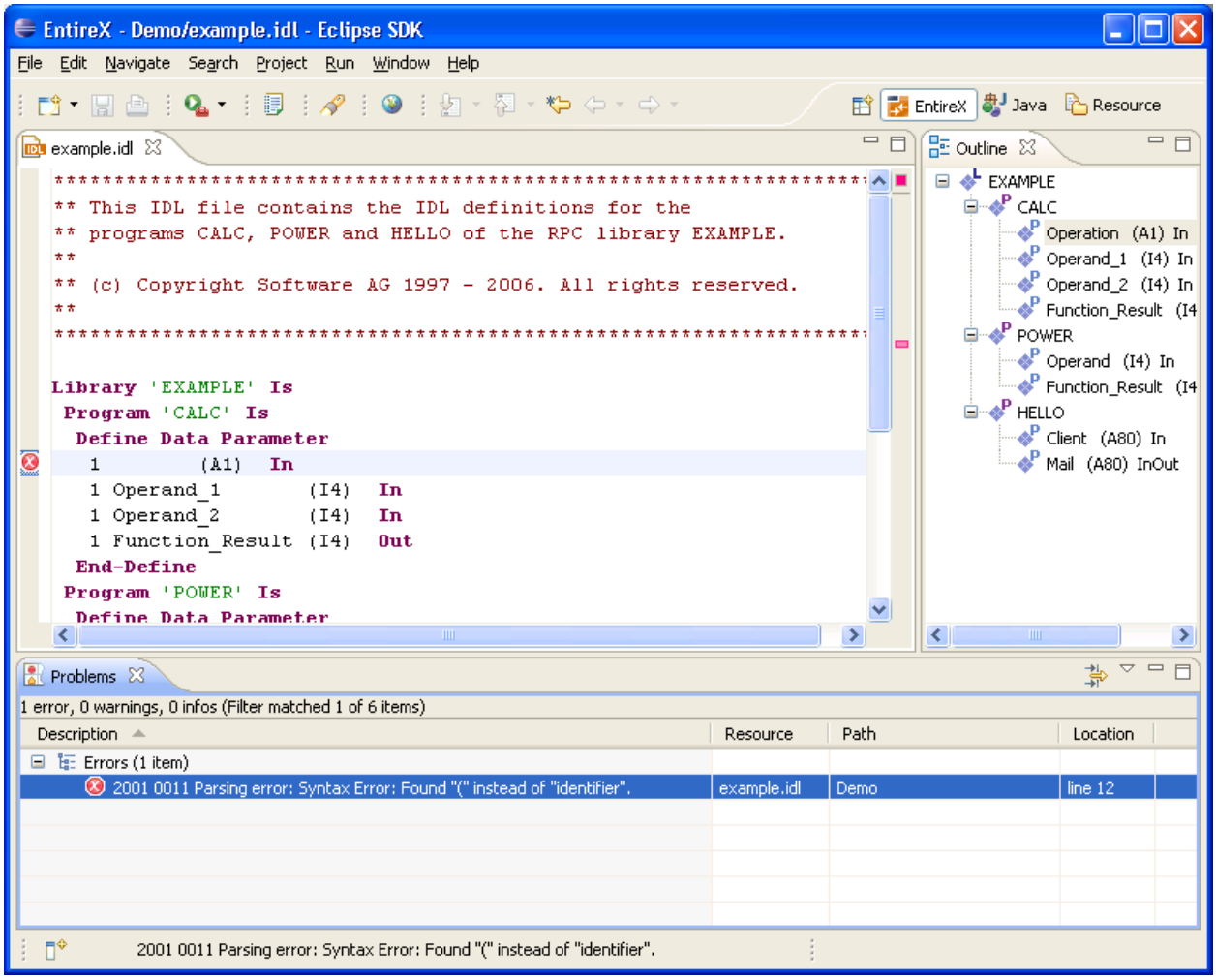
Property	Description
Name	Name of the library or program.
Preceding comment lines	Any comment in a preceding line or lines that applies to this line is displayed here. You can also add here a comment to this library or program in a definition line.
Same-line comment	Any comment in the current line is displayed here. You can also add a comment to the current line.

Parameter

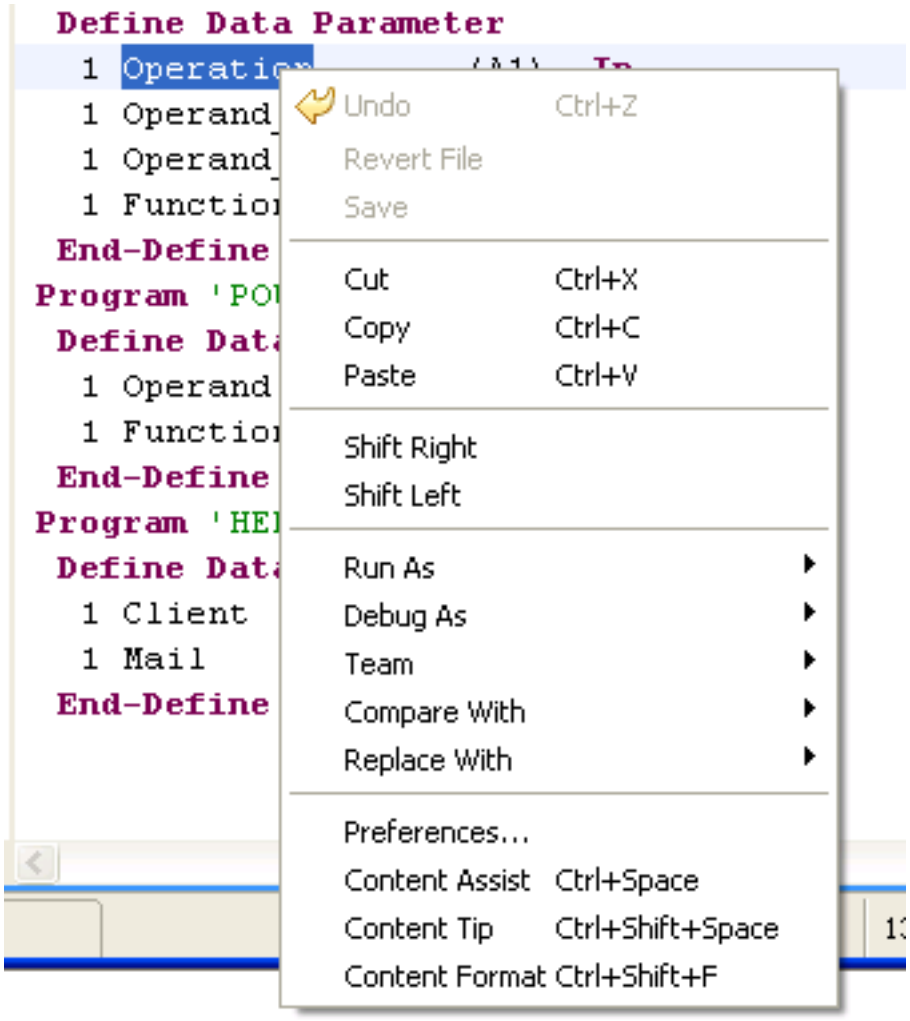
Property	Description
Aligned	Specifies whether the current line contains an `aligned` statement.
Data Type	Here you can select data types from a drop-down list.
Dimension 1	First array dimension.
Dimension 2	Second array dimension.
Dimension 3	Third array dimension.
Name	Name of the parameter.
Preceding comment lines	Any comment in a preceding line or lines that applies to this line is displayed here. You can also add here a comment to this parameter.
Same-line comment	Any comment in the current line is displayed here. You can also add a comment to the current line.
Size and precision	Size and precision of the parameter.
Structure reference	If a structure is referenced, its name is displayed here.

IDL Editor Problems View

The Problems view documents syntax errors in the current IDL file, depending on the filter settings of the view. Only a document that is free of syntax errors can be displayed in the Outline view. Additionally, the line containing the error is marked with an error symbol and flagged to the right of the scrollbar.


EntireX - Demo/example.idl - Eclipse SDK
File Edit Navigate Search Project Run Window Help
EntireX Java Resource
example.idl
** This IDL file contains the IDL definitions for the
** programs CALC, POWER and HELLO of the RPC library EXAMPLE.
** (c) Copyright Software AG 1997 - 2006. All rights reserved.
Library 'EXAMPLE' Is
Program 'CALC' Is
Define Data Parameter
1 (A1) In
1 Operand_1 (I4) In
1 Operand_2 (I4) In
1 Function_Result (I4) Out
End-Define
Program 'POWER' Is
Define Data Parameter
Outline
EXAMPLE
CALC
Operation (A1) In
Operand_1 (I4) In
Operand_2 (I4) In
Function_Result (I4
POWER
Operand (I4) In
Function_Result (I4
HELLO
Client (A80) In
Mail (A80) InOut
Problems
1 error, 0 warnings, 0 infos (Filter matched 1 of 6 items)
Description Resource Path Location
Errors (1 item)
2001 0011 Parsing error: Syntax Error: Found "(" instead of "identifier". example.idl Demo line 12
2001 0011 Parsing error: Syntax Error: Found "(" instead of "identifier".

Context Menu of the IDL Editor

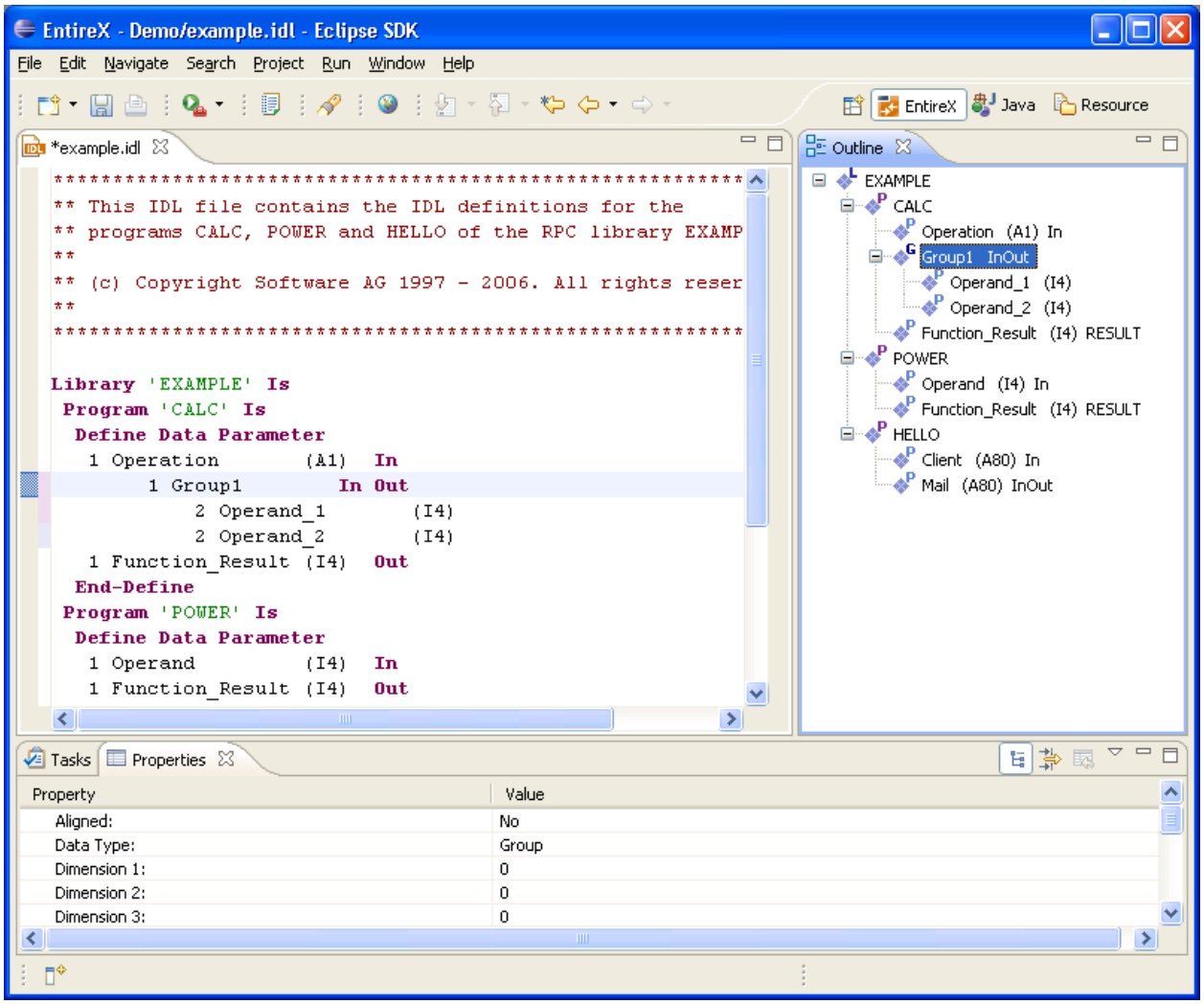


In addition to standard editor commands such as `Undo`, `Revert File`, `Cut`, `Copy`, `Paste`, `Save` etc., the context menu of the IDL Editor view offers two additional commands for shifting an element in the text: `Shift Right` to add a tab, and `Shift Left` to remove a tab.

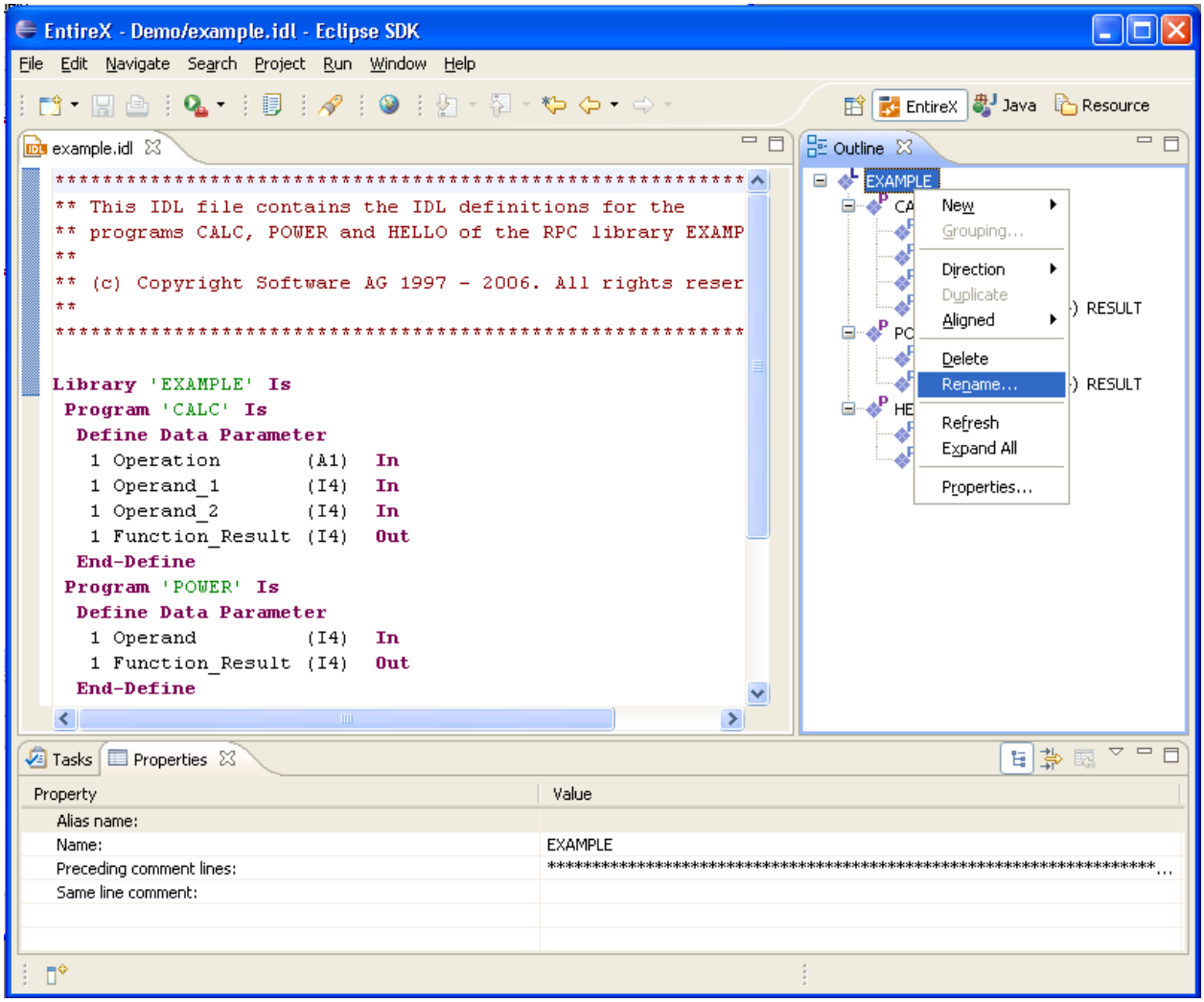
Editing an IDL File from the Outline View

This example describes the operation of the IDL file outline context menu. These operations allow you to insert new IDL syntax elements without detailed knowledge of the syntax.

In this example we generate an IDL file by using the Software AG IDL File Wizard. The **Outline** view is opened (if it is not open already).



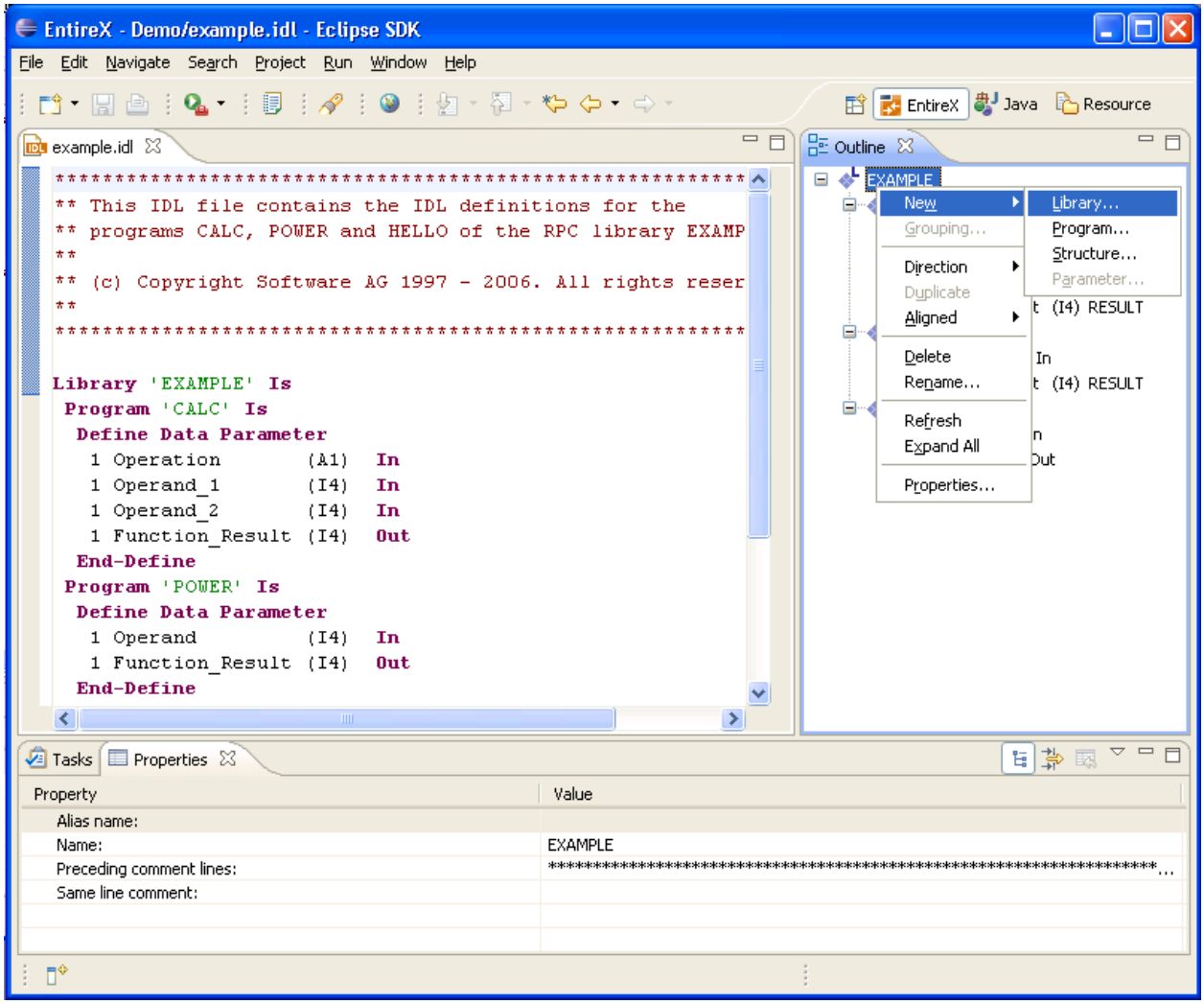
We can change the name of the library for example by selecting the library and choosing **Rename** in the context menu. The same applies to program and parameters.



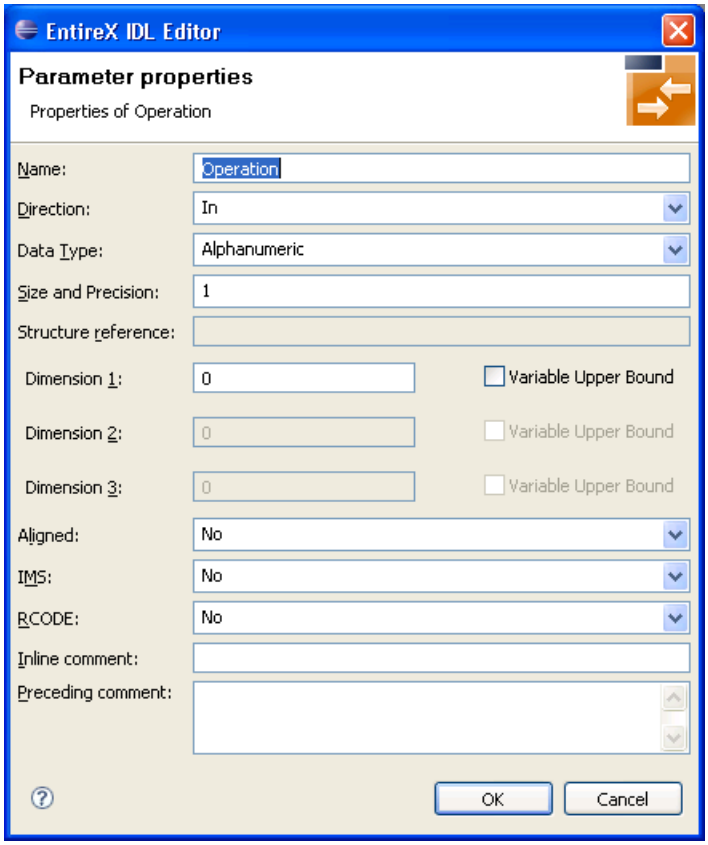
Adding a New Element

Choose the **New...** submenu of the context menu. To add an additional library to the IDL file, choose **Library**.

A **Library Properties** dialog is displayed, where you can enter the name of the library, the alias and comments.



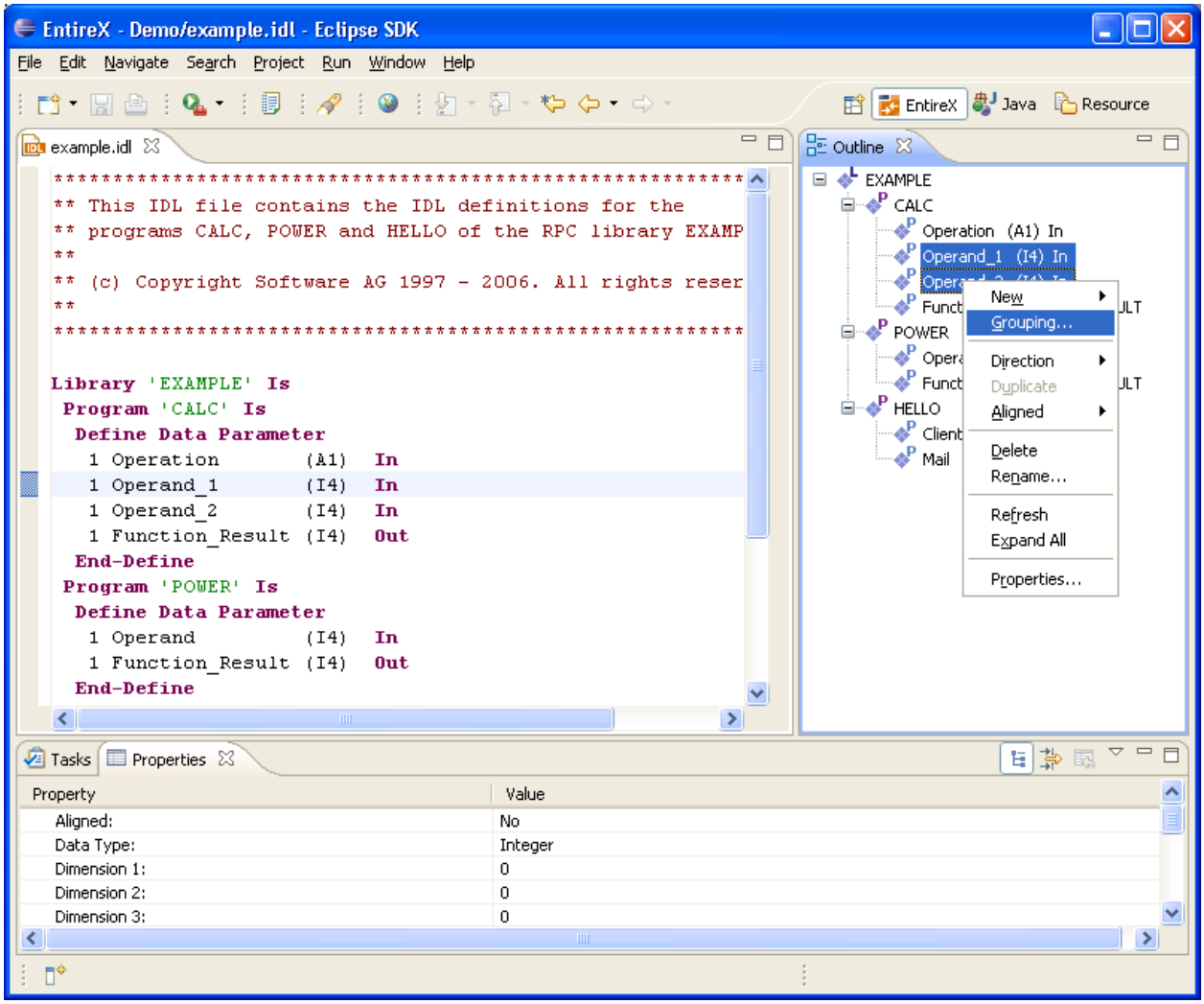
Select a parameter and choose **Properties** in the context menu. In the **Parameter properties** dialog you can set all required properties.

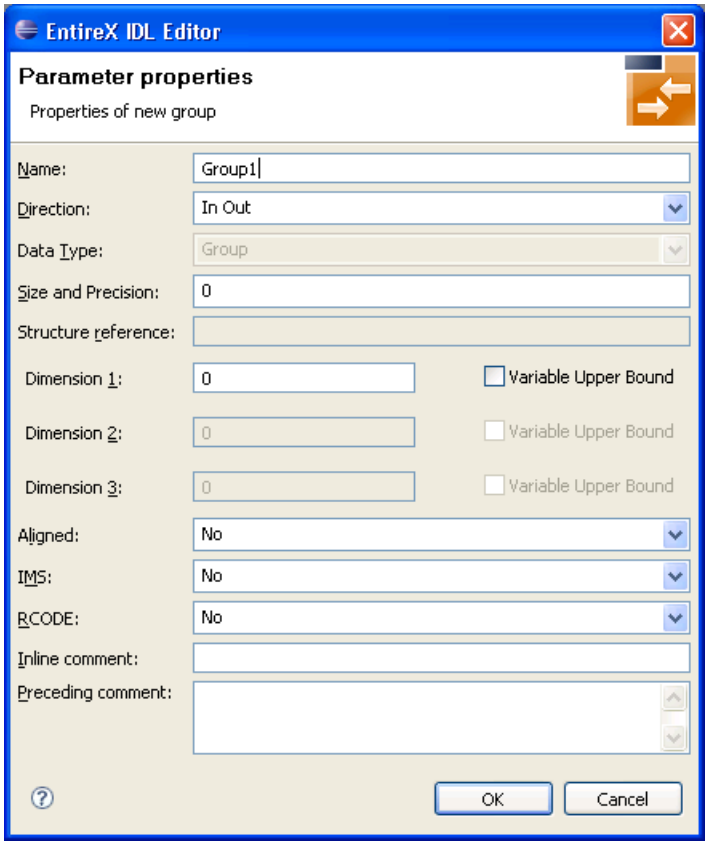


The properties of libraries, programs and structures are changed in a similar way.

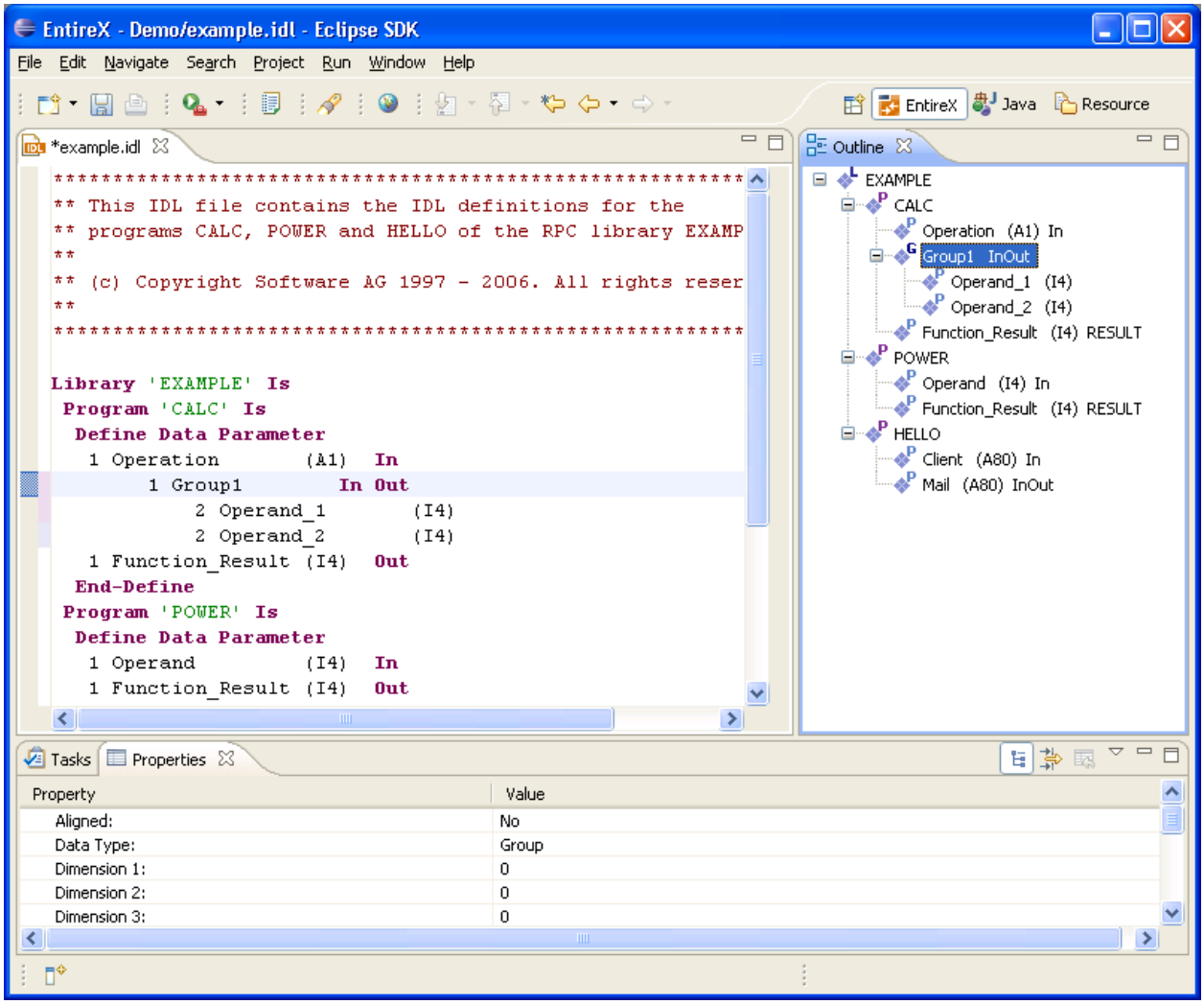
Grouping

To group these two parameters, select both parameters and choose **Grouping** in the context menu. This inserts a new group into the program. The new group contains the two parameters.


EntireX - Demo/example.idl - Eclipse SDK
File Edit Navigate Search Project Run Window Help
EntireX Java Resource
example.idl
** This IDL file contains the IDL definitions for the
** programs CALC, POWER and HELLO of the RPC library EXAMP
** (c) Copyright Software AG 1997 - 2006. All rights reser
Library 'EXAMPLE' Is
Program 'CALC' Is
Define Data Parameter
1 Operation (A1) In
1 Operand_1 (I4) In
1 Operand_2 (I4) In
1 Function_Result (I4) Out
End-Define
Program 'POWER' Is
Define Data Parameter
1 Operand (I4) In
1 Function_Result (I4) Out
End-Define
Outline
EXAMPLE
CALC
Operation (A1) In
Operand_1 (I4) In
POWER
HELLO
Client
Mail
New
Grouping...
Direction
Duplicate
Aligned
Delete
Rename...
Refresh
Expand All
Properties...
Tasks
Properties
Property
Value
Aligned: No
Data Type: Integer
Dimension 1: 0
Dimension 2: 0
Dimension 3: 0



In the **Parameter Properties** dialog box you can enter all the settings required for this group.


EntireX - Demo/example.idl - Eclipse SDK
File Edit Navigate Search Project Run Window Help
EntireX Java Resource
*example.idl
** This IDL file contains the IDL definitions for the
** programs CALC, POWER and HELLO of the RPC library EXAMP
** (c) Copyright Software AG 1997 - 2006. All rights reser
Library 'EXAMPLE' Is
Program 'CALC' Is
Define Data Parameter
1 Operation (A1) In
1 Group1 In Out
2 Operand_1 (I4)
2 Operand_2 (I4)
1 Function_Result (I4) Out
End-Define
Program 'POWER' Is
Define Data Parameter
1 Operand (I4) In
1 Function_Result (I4) Out
Outline
EXAMPLE
CALC
Operation (A1) In
Group1 InOut
Operand_1 (I4)
Operand_2 (I4)
Function_Result (I4) RESULT
POWER
Operand (I4) In
Function_Result (I4) RESULT
HELLO
Client (A80) In
Mail (A80) InOut
Tasks
Properties
Property
Value
Aligned: No
Data Type: Group
Dimension 1: 0
Dimension 2: 0
Dimension 3: 0

3 Software AG IDL File

A Software AG IDL file contains definitions of the interface between client and server. The IDL file is used by Software AG wrappers to generate RPC clients, RPC servers and tester etc. on the basis of these definitions. The IDL file can be edited by the IDL Editor provided by plug-ins for Eclipse.

This document contains a descriptive introduction to IDL files. The syntax of IDL files in a formal notation is given under ***Software AG IDL Grammar***.

Introduction to the IDL File

The IDL's syntax looks similar to a Software AG Natural parameter data definition statement.

```
Library 'EXAMPLE' Is
        Program 'CALC' Is
                Define Data Parameter
                1 Operator         (A1) In
                1 Operand_1        (I4) In
                1 Operand_2        (I4) In
                1 Function_Result  (I4) Out
                End-Define
```

The syntax is described in a formal notation under ***Software AG IDL Grammar***.

IDL Data Types

The table below uses the following metasymbols and informal terms for the IDL.

- The metasymbols [and] surround optional lexical entities.
- The informal term *number* (or in some cases *number*.*number*) is a sequence of numeric characters, for example 123.

Type and Length	Description	Example	See Notes
A*number*	Alphanumeric	A100	1, 2, 7, 17, 20
AV	Alphanumeric variable length	AV	1, 2, 7, 17, 20, 21
AV*number*	Alphanumeric variable length with maximum length	AV100	1, 2, 7, 17, 20, 21
B*number*	Binary	B10	1, 2, 15
BV	Binary variable length	BV	1, 2, 15, 21
BV*number*	Binary variable length with maximum length	BV128	1, 2, 15, 21
D	Date	D	3, 4, 13

Type and Length	Description	Example	See Notes
F4	Floating point (small)	F4	11, 13, 16
F8	Floating point (large)	F8	12, 13, 16
I1	Integer (small)	I1	8
I2	Integer (medium)	I2	9
I4	Integer (large)	I4	10
K*number*	Kanji	K20	1, 2, 7, 17, 18, 20
KV	Kanji variable length	KV	1, 2, 7, 17, 18, 20, 21
KV*number*	Kanji variable length with maximum length	KV200	1, 2, 7, 17, 18, 20, 21
L	Logical	L	3, 14
N*number*[. *number*]	Unpacked decimal	N8 or N8.2	6, 13
NU*number*[. *number*]	Unpacked decimal unsigned	NU2 or NU6.2	6, 13
P*number*[. *number*]	Packed decimal	P12 or P10.3	6, 13
PU*number*[. *number*]	Packed decimal unsigned	PU3 or PU4.2	6, 13
T	Time	T	3, 5, 13
U*number*	Unicode	U100	2, 19
UV	Unicode variable length	UV	2, 19, 21
UV*number*	Unicode variable length with maximum length	UV200	2, 19, 21

Note that equivalents of the data types are not necessarily supported in every target programming language environment. Also, value ranges of the mapped data type can differ. See *Mapping Software AG IDL Data Types* in the respective Wrapper or language-specific documentation.

Notes:

1. There is, however, an absolute limit (1 GB) which cannot be exceeded.
2. The maximum length you can specify depends on your hardware and software configuration (apart from this product).
3. The length is implicit and must not be specified.
4. The supported range is from 1.1.0001 up to 31.12.9999. Dates BC (before the birth of Christ) are not supported.

   It is also possible to transfer 1.1.0000 as a value. This is a special value (because there is no year 0) and denotes “no date” is given. The no date value is the internal state of a `#DATE` variable (Natural type D) after a `RESET #DATE` is executed within Natural programs. The target language environment determines how 'no date' is handled.

   See the notes under data type D in the section *Mapping Software AG IDL Data Types* to the target language environment C | Java | .NET.
5. The data type T has two different meanings:

- A time-only meaning, which transfers a time without a date. The time-only meaning always uses the invalid date 1.1.000 for the date part. The time part has a value range from 00:00:00.0 to 23:59:59.9. This time-only meaning is not supported.
- A timestamp meaning, consisting of a date and time.

  The supported range is from 1.1.0001 0:00:00.0 up to 31.12.9999 23:59:59.9. Dates BC (before the birth of Christ) are not supported.

  It is also possible to transfer 1.1.0000 0:00:00.0 as a value. This is a special value (because there is no year 0) and denotes "no time" is given. The "no time" value is the internal state of a `#TIME` (Natural type T) variable after a `RESET #TIME` is executed within Natural programs. The target language environment determines how "no time" is handled.

  See the notes under data type T in the section *Mapping Software AG IDL Data Types* to the target language C | Java | .NET.

6. The term *number*[. *number*] describes the number as it is: The first number is the number of digits before the decimal point and the second number is the number of digits after the decimal point. The total number of digits (before and after the decimal point) must not exceed 29. The number of digits after the decimal point must not exceed 7.
7. The length is given in bytes, not in number of characters.
8. The valid integer range is from -128 up to +127.
9. The valid integer range is from -32768 up to +32767.
10. The valid integer range is from -2147483648 up to +2147483647.
11. The following term restricts the valid range which can be transferred from -n.nnnnnn+Enn up to +n.nnnnnn+Enn. A mantissa of 7 decimal digits and an exponent of 2 decimal digits.
12. The following term restricts the valid range which can be transferred from -n.nnnnnnnnnnnnnnn+Enn up to +n.nnnnnnnnnnnnnnn+Enn. A mantissa of 16 decimal digits and an exponent of 2 decimal digits.
13. The real valid range and precision can be restricted by the mapping to the target language environment.
14. Valid values are TRUE and FALSE.
15. The length is given in bytes.
16. When using floating-point values, rounding errors can occur when converting to the target language environment. Thus, values from sender and receiver might differ slightly.
17. In environments where multibyte, double-byte or other complex codepages are used, alphanumeric data may increase or decrease during conversion. Thus, to match the field length restriction given by the IDL types A and AV with maximum length, data must be truncated, otherwise unpredictable results will occur. The most popular internationalization approach *ICU Conversion* under *Introduction to Internationalization* with `CONVERSION=SAGTRPC` takes care of data increase/decrease.

We recommend always using SAGTRPC for RPC data streams. *Conversion with Multibyte, Double-byte and other Complex Codepages* will always be correct, and *Conversion with Single-byte Codepages* is also efficient because SAGTRPC detects single-byte codepages automatically. See *Conversion Details*.

See also *Configuring ICU Conversion* under *Configuring Broker for Internationalization* in the platform-specific administration documentation.

18. In environments that use EBCDIC stateful codepages, encoded with escape technique (SI/SO bytes), and where the most popular internationalization approach *ICU Conversion* under *Introduction to Internationalization* with `CONVERSION=SAGTRPC` is used, the IDL types K and KV fields allow you to transfer double-byte data without SO and SI bytes. This feature is designed for use in Asian countries. For more information see *Conversion with Multibyte, Double-byte and other Complex Codepages*.
19. The length is given in 2-byte Unicode code units following the Unicode standard. UTF-16. The maximum length is restricted to 805306367 2-byte code units.

    Depending on your target environment and target programming language, the mapping may follow a different Unicode standard, for example UTF-32.
20. If *SAGTRPC User Exit* under *Introduction to Internationalization* is used as the internationalization approach, the handling of the different IDL types depends on the implementation of the SAGTRPC user exit. This is your responsibility as user. See *Writing SAGTRPC User Exits* in the platform-specific administration documentation.
21. Variable-length (e.g. AV, AV*n*) fields are transferred in the RPC data stream in the length specified. A defined maximum in the IDL file limits the length that can be transferred.

    Variable-length fields with maximum (e.g. AV*n*) are important for connections to endpoints that have no concept of variable-length data, such as COBOL (see *Software AG IDL to COBOL Mapping*) and PL/I (see *Software AG IDL to PL/I Mapping*).

Fixed and Unbounded Arrays

A fixed array is transferred in the RPC data stream with all its elements.

With an unbounded array, the current number of elements and their contents are transferred in the RPC data stream. A defined maximum in the IDL file limits the number of elements that can be transferred.

For the formal syntax of arrays, refer to *`array-definition`* under ***Software AG IDL Grammar***.

Unbounded arrays with a maximum are important for connections to COBOL, which supports a similar concept with the `OCCURS DEPENDING ON` clause. See *Tables with Variable Size - `DEPENDING ON` Clause* under *COBOL to IDL Mapping* in the IDL Extractor for COBOL documentation.

Rules for Coding IDL Files

1. Statements and their lexical entities can begin in any column and are separated by any number of whitespace characters: blank, new line carriage return, horizontal tab, and form feed.
2. The maximum line length allowed in an IDL file is 256 characters.
3. Comments can be entered in the following ways:
   - If the entire line is to be used for a user comment, enter an asterisk or a slash and an asterisk in columns 1 and 2 of the line:

     ```
     *    USER COMMENT
     /*   USER COMMENT
     ```

   - If only the latter part of a line is to be used for a user comment, enter an asterisk or slash asterisk.

     ```
       1 NAME    (A20)      * USER COMMENT
       1 NUMBER  (A15)     /* USER COMMENT
     ```

Rules for Coding Group and Parameter Names

Group and parameter names

1. can be defined with the following characters:
   - characters: a to z
   - characters: A to Z
   - digits: 0 to 9 (a digit must not be the first character)
   - special characters: - _ $ # & @ + /

   other characters are not allowed.
2. are limited to a maximum length of 31 characters
3. are not allowed to be the same as a valid type-length specification.

   For example:

```
1 P1 (P1) In Out
```

is invalid and will cause an error because the name `P1` is identical to the type-length `P1`.

4. must adhere to the rules of the target programming language, for example to permitted special characters or reserved keywords.
5. cannot be defined as the following reserved names:

   `ALIGNED`, `CALLNAT`, `DATA`, `DEFINE`, `END-DEFINE`, `IMS`, `IN`, `INOUT`, `IS`, `LIBRARY`, `OUT`, `PARAMETER`, `PROGRAM`, `RCODE`, `STRUCT`, `VERSION`.

6. must be unique and must not conflict with those of the target programming language, see the following portion of an IDL file

   ```
   Define Data Parameter
   1 AA   (I2)
   1 AA   (I4)
   1 long (I4)
   End-define
   ```

   and the output generated with the client.tpl as the template for target language C:

   ```
   short int AA;
   long      AA;      /*erroneous, double declaration*/
   long      long;    /*erroneous, double declaration*/
   ```

   The ambiguous declaration of `AA` and `long` is passed unchecked and the stub will be generated. As you can see, this is not valid C syntax.

Rules for Coding Library, Library Alias, Program, Program Alias and Structure Names

The following rules apply to library, library alias, program, program alias and structure names:

1. Names are restricted by length. Library, library alias, program and program alias are restricted to a maximum length of 128 characters. A structure name is restricted to a maximum length of 31 characters.
2. Names must adhere to the rules of the target programming language, for example regarding permitted special characters or reserved keywords.
3. Names should not start with the prefix "`SAG`". The prefix "`SAG`" is used within the delivered IDL files. See *Change RPC Password by Wrappers and RPC Clients* and *Command and Info Services IDLs* under RPC Programming for more information.

4. Names must be unique and different within the IDL file after conversion of the name to lowercase or uppercase characters. You cannot use the same name for a library, library alias, program, program alias and structure

   Example: The following names are not allowed within an IDL file:

   - `MYLIBRARY` and `MyLibrary`
   - `CALC` and `Calc`
   - `MYSTRUCTURE` and `mystructure`

4 Software AG IDL Grammar

A Software AG IDL file contains definitions of the interface between client and server. The IDL file is used by Software AG wrappers to generate RPC clients, RPC servers and tester etc. on the basis of these definitions. The IDL file can be edited by the IDL Editor provided by plug-ins for Eclipse.

This chapter explains the syntax of IDL files in a formal notation. A more descriptive introduction to IDL files is given in the chapter *Software AG IDL File* in the IDL Editor documentation. This chapter covers the following topics:

Meta Definitions

The following metasymbols are used to define the IDL:

- The metasymbols [and] surround optional lexical entities.
- The metasymbols { and } surround optional lexical entities which may be repeated a number of times.
- The metasymbol ::= separates the lexical entity to be defined (on the left) from the definition (on the right).
- The metasymbol | separates lexical entities on the definition side (on the right) meaning all terms are valid to define the lexical entity to be defined (on the left).

The following basic terms are used to describe the IDL:

- The informal term `number` is a sequence of numeric digits e.g. 123.
- The informal term `string` is a sequence of characters. It can contain any character except enclosing apostrophes.

  Examples are: `'DARMSTADT' '#FRANKFURT' '&MUNICH'`.
- Any terms in uppercase, special characters such as apostrophe, colon (other than the metasymbols above) are terminal lexical entities and must be entered as is.
- The term `identifier` is a sequence of
  - characters: a to z
  - characters: A to Z
  - digits: 0 to 9 (a digit must not be the first character)
  - special characters: - _ $ # & @ + /

Syntax of the IDL File

Syntax

Software AG IDL	::= library-definition { library-definition }

Description

- The IDL may contain any number of library-definitions.
- One library-definition must be contained in an IDL file.

library-definition

A library-definition is the grouping of servers (remote procedures).

Syntax

library-definition	::= LIBRARY 'library-name' [:'library-alias'] IS { interface }
library-name	::= string
library-alias	::= string
interface	::= program-definition | structure-definition

Description

library-definition	A library-definition is valid until the next library-definition or end of file.
library-name	■ The library-name is used to generate RPC components. How this takes place (e.g. to form a source file name, class name etc.) depends on the target programming language. The wrappers will adapt the library-name to the requirements of the target programming language when special characters occur in the library-name. See *Mapping Library Name and Alias* in the respective Wrapper documentation. ■ The library-name is also sent (without modifying any special characters) from the RPC client component to the RPC server. In the RPC server the library-name may be used to locate the target server. See *Locating and Calling the Target Server* in the platform-specific administration or RPC server documentation.

	■ Certain rules apply to `library-name`. See ***Rules for Coding Library, Library Alias, Program, Program Alias and Structure Names***.
`library-alias`	■ Alias of the `library-name`. ■ The purpose of an alias is to allow a different name on the RPC client side from the name on the RPC server side. This is helpful when integrating a system with a short name (e.g. CICS, z/OS, Natural, where up to 8 characters are allowed, or IBM i, where up to 10 characters are allowed) on one side and on the other side an environment with fewer restrictions (UNIX, Windows, Java). ■ The `library-alias` may be used as a name in the target programming language to form the generated RPC client and RPC server components instead of the `library-name`. How the `library-alias` is used to generate components (e.g. to form a source file name, class name etc.) depends on the target programming language. See *Mapping Library Name and Alias* in the respective Wrapper documentation. ■ The `library-alias` is always used as is (it is not adapted by the IDL Editor and the Workbench wrappers when special characters occur within the `library-alias` as it is for the `library-name`), i.e. the user is responsible for a valid target programming language name. ■ The `library-alias` is not sent to the target RPC server. The `library-name` is always sent instead. ■ Certain rules apply to `library-name`. See ***Rules for Coding Library, Library Alias, Program, Program Alias and Structure Names***.
`interface`	■ A `program-definition` or `structure-definition` concludes the `library-definition`. ■ Any number of `program-definitions` or `structure-definitions` can be embedded in a `library-definition`.

Example (without alias usage)

```
Library 'ServerLibrary' Is ..
```

Example (with alias usage)

```
Library 'ServerLibrary': 'AliasServerLibrary' Is ..
```

program-definition

A `program-definition` describes the parameters of servers (remote procedures).

Syntax

`program-definition`	`::= PROGRAM 'program-name' [:'program-alias'] IS parameter-data-definition`
`program-name`	`::= string`
`program-alias`	`::= string`

Description

`program-definition`	■ A `program-definition` is valid until the next `program-definition`, `structure-definition`, `library-definition` or end of file. ■ Any `program-definition` must be embedded in a `library-definition`. ■ Any number of`program-definitions`can be embedded in a `library-definition`. ■ One `parameter-data-definition` must conclude the `program-definition`.
`program-name`	■ The `program-name` is used to generate RPC components. How this takes place (e.g. how to form a source file name, method, function or program name etc.) depends on the target programming language. The IDL Editor and the integrated Workbench wrappers will adapt the `program-name` to the requirements of the target programming language when special characters occur in the `program-name`. See *Mapping Program Name and Alias* in the respective Wrapper documentation. ■ The `program-name` is also sent (without modifying any special characters) from the RPC client component to the RPC server. In the RPC server the `program-name` is used to locate the target server. See *Locating and Calling the Target Server* in the platform-specific administration or RPC server documentation. ■ Certain rules apply to `program-name`. See ***Rules for Coding Library, Library Alias, Program, Program Alias and Structure Names***.
`program-alias`	■ Alias of the `program-name`. ■ The purpose of an alias is to allow a different name on the RPC client side from the name on the RPC server side. This is helpful when integrating a system with a short name (e.g. CICS, z/OS, Natural, where up to 8 characters are allowed, or

	IBM i, where up to 10 characters are allowed) on one side and on the other side an environment with fewer restrictions (Window, UNIX, Java). ■ The `program-alias` may be used as a name in the target programming language to form the generated RPC client and RPC server components instead of the `program-name`. How the `program-alias` is used to generate components (e.g. to form a source file name, method, function or program name etc.) depends on the target programming language. See *Mapping Program Name and Alias* in the respective Wrapper documentation. ■ The `program-alias` is always used as is (it is not adapted by the IDL Editor and the wrappers when special characters occur within the `program-alias` as it is for the `program-name`), i.e. the user is responsible for a valid target programming language name. ■ The `program-alias` is not sent to the target server. The `program-name` is always sent instead. ■ Certain rules apply to `program-alias`. See ***Rules for Coding Library, Library Alias, Program, Program Alias and Structure Names***.

Example (without alias usage):

```
Library 'ServerLibrary' Is
    Program 'ServerName' Is ..
```

Example (with alias usage):

```
Library 'ServerLibrary': 'AliasServerLibrary' Is
       Program 'ServerName' : 'AliasServerName' Is ..
```

structure-definition

A `structure-definition` describes a user-defined type for reusability, referenced in a ***`structure-parameter-definition` (IDL)***.

Syntax

`structure-definition`	`::= STRUCT 'structure-name' IS parameter-data-definition`
`structure-name`	`::= string`

Description

`structure-definition`	■ A `structure-definition` is valid until the next `program-definition`, `structure-definition`, `library-definition` or end of file. ■ Any `structure-definition` must be embedded in a `library-definition`. ■ Any number of `structure-definitions` can be embedded in a `library-definition`. ■ One `parameter-data-definition` must conclude the `structure-definition`. ■ Structures are mapped to various concepts depending on the target programming language. See *Mapping Structures* in the respective Wrapper documentation.
`structure-name`	■ The `structure-name` used for reference. ■ Certain rules apply to `structure-name`. See ***Rules for Coding Library, Library Alias, Program, Program Alias and Structure Names***.

Example

```
Library 'ServerLibrary': 'AliasServerLibrary' Is
Struct 'Person' Is ..
```

parameter-data-definition

The `parameter-data-definition` describes the parameters of a server when it is embedded in a `program-definition`. It describes a user-defined type when it is embedded in a `structure-definition`.

Syntax

parameter-data-definition	::= DEFINE DATA PARAMETER simple-parameter-definition | group-parameter-definition | structure-parameter-definition { simple-parameter-definition | group-parameter-definition | structure-parameter-definition } END-DEFINE

Description

parameter-data-definition	■ The parameter-data-definition consists of a starting token sequence (Define Data Parameter) and an ending token sequence (End-Define). ■ Any number of simple-parameter-definition, group-parameter-definition and structure-parameter-definition can be embedded in a parameter-data-definition. ■ At least one simple-parameter-definition, group-parameter-definition or structure-parameter-definition must exist (at level 1) without the *attribute-list*. ■ The parameter-data-definition must conclude the corresponding program-definition or structure-definition. ■ There can only be one parameter-data-definition for each program-definition or structure-definition.

Example of a Program:

```
Library 'ServerLibrary' Is
        Program 'ServerName' Is ..
                Define Data Parameter
                ...
                End-Define
```

Example of a Structure:

```
Library 'ServerLibrary': 'AliasServerLibrary' Is
         Struct 'Person' Is ..
                Define Data Parameter
                ...
                End-Define
```

simple-parameter-definition

The construct `simple-parameter-definition` describes the syntax of a simple parameter, i.e. not a group (groups are described in a `group-parameter-definition`), not a reference to a structure (referencing a structure is described in *`structure-parameter-definition` (IDL)*).

Syntax

`simple-parameter-definition`	`::= level parameter-name (type-length[/array-definition]) [attribute-list]`
`level`	`::= number`
`parameter-name`	`::= identifier`
`type-length`	See *IDL Data Types*.

Description

`level`	■ Level number is a 1 or 2-digit number in the range from 01 to 99 (the leading 0 is optional) used in conjunction with parameter grouping. ■ Parameters assigned a level number of 02 or greater are considered to be members of the immediately preceding group that has been assigned a lower level number. ■ Do not skip level numbers when assigning the level numbers for group members.
`parameter-name`	■ The name of the parameter. The `parameter-name` is used as name in the target programming language. It is adapted by the IDL Editor and the wrappers to the requirements of the target programming language. See *Mapping Parameter Names* in the respective Wrapper documentation. ■ Certain rules apply to `parameter-name`. See *Rules for Coding Group and Parameter Names*.
`type-length`	The type and length of the parameter. See *IDL Data Types*.

Example

```
...
  1 PERSON-ID (N10)
  1 PERSON-NAME (A100)
...
```

group-parameter-definition

The construct `group-parameter-definition` describes the syntax of a group.

Syntax

`group-parameter-definition`	`::= level group-name [(/array-definition)] [attribute-list]`
`level`	`::= number`
`group-name`	`::= identifier`

Description

`level`	See *`simple-parameter-definition`*.
`group-name`	■ The name of the group. ■ The definition of a group enables references to a series of parameters (can also be only 1 parameter) by using the group name. This provides a convenient and efficient method of referencing a series of consecutive parameters. ■ A group may contain other groups, structures (`structure-parameter-definition`) or parameters (`simple-parameter-definition`) as group members. ■ Certain rules apply to `group-name`. See ***Rules for Coding Group and Parameter Names***. ■ Groups are mapped to various concepts depending on the target programming language. See *Mapping Groups and Periodic Groups* in the respective Wrapper documentation.

Example

```
...
1 PERSON /* this is the group */
2 PERSON-ID (N10) /* this is a group member */
2 PERSON-NAME (A100) /* this is also a group member */
...
```

structure-parameter-definition (IDL)

The construct `structure-parameter-definition` describes the syntax of a reference to a structure.

Syntax

`structure-parameter-definition`	`::= level parameter-name (structure-reference[/array-definition]) [attribute-list]`
`level`	`::= number`
`parameter-name`	`::= identifier`
`structure-reference`	`::= 'structure-name'`

Description

`level`	See *`simple-parameter-definition`*.
`parameter-name`	See *`simple-parameter-definition`*.
`structure-reference`	■ `structure-name` of the referenced structure. ■ The referenced `structure-name` must be surrounded by quotation marks. ■ Structures are mapped to various concepts depending on the target programming language. See *Mapping Structures* in the respective Wrapper documentation. ■ Certain rules apply to `structure-name`. See ***Rules for Coding Library, Library Alias, Program, Program Alias and Structure Names***.

Example

```
STRUCT 'Person' Is /* this defines the structure person */
Define Data Parameter
1 PERSON
2 PERSON-ID (N10)
2 PERSON-NAME (A100)
End-Define
...
1 FATHER ('Person') /* this references the structure */
1 MOTHER ('Person') /* this references the structure */
1 CHILDS ('Person'/10) /* this references the structure */
...
```

array-definition

Arrays can have either fixed upper bounds or variable upper bounds, so-called unbounded arrays.

Syntax

array-definition	::= fixed-bound-array | unbounded-array
fixed-bound-array	::= [fixed-bound-array-index [,fixed-bound-array-index [,fixed-bound-array-index]]]
unbounded-array	::= [unbounded-array-index [,unbounded-array-index [,unbounded-array-index]]]
fixed-bound-array-index	::= [lower-bound:] upper-bound
unbounded-array-index	::= [1:] V[maximum-upper-bound]
lower-bound	::= number
upper-bound	::= number
maximum-upper-bound	::= number

Description

array-definition	■ Arrays with a fixed size of elements are fixed bound arrays. ■ Arrays with a variable number of elements are so called unbounded arrays. ■ Arrays are one, two or three-dimensional.
fixed-bound-array	■ Almost all programming languages have a concept of arrays with a fixed size of elements. ■ See *Mapping Fixed and Unbounded Arrays* in the respective Wrapper documentation.

`unbounded-array`	■ Unbounded arrays are not supported by all endpoints, or are supported with restrictions. ■ See *Mapping Fixed and Unbounded Arrays* in the respective Wrapper documentation.
`fixed-bound-array-index`	■ If an `array-index` is of the format `[lower-bound:]upper-bound` it describes an array with fixed bounds. ■ The fixed number of occurrences is calculated as `upper-bound` – `lower-bound` + 1.
`unbounded-array-index`	■ If `maximum-upper-bound` is given, the variable number of occurences (elements) can vary between 0 (empty unbounded array) and `maximum-upper-bound`. ■ If no `maximum-upper-bound` is given, there is no practical upper limit on the variable number of occurences.(1) ■ Empty unbounded arrays with zero number of occurences are always possible. The optional notation [1:] is ignored and has no effect. **Note:** (1) The implementation limits the number to 2,147,483,647 elements. ■ If an `array-index` is of the format `[1:]V[maximum-upper-bound]` it describes an unbounded array with variable bounds.
`lower-bound`	■ The `lower-bound` value is optional. ■ If the `lower-bound` is not given, the default is 1.
`upper-bound`	■ The `upper-bound` value must be entered. ■ The `upper-bound` value must be greater than or equal to the `lower-bound`.
`maximum-upper-bound`	■ The `maximum-upper-bound` value is optional. ■ It defines a limit which cannot be exceeded.

Example of Arrays with Fixed Bounds

```
...
1 NAMES (A100/10) /* 1 dimensional array */
1 TUPLES (A100/10,10) /* 2 dimensional array */
1 TRIPLES (I1/1:20,1:20,1:20) /* 3 dimensional array */
...
```

Example of Arrays with Variable Upper-bounds

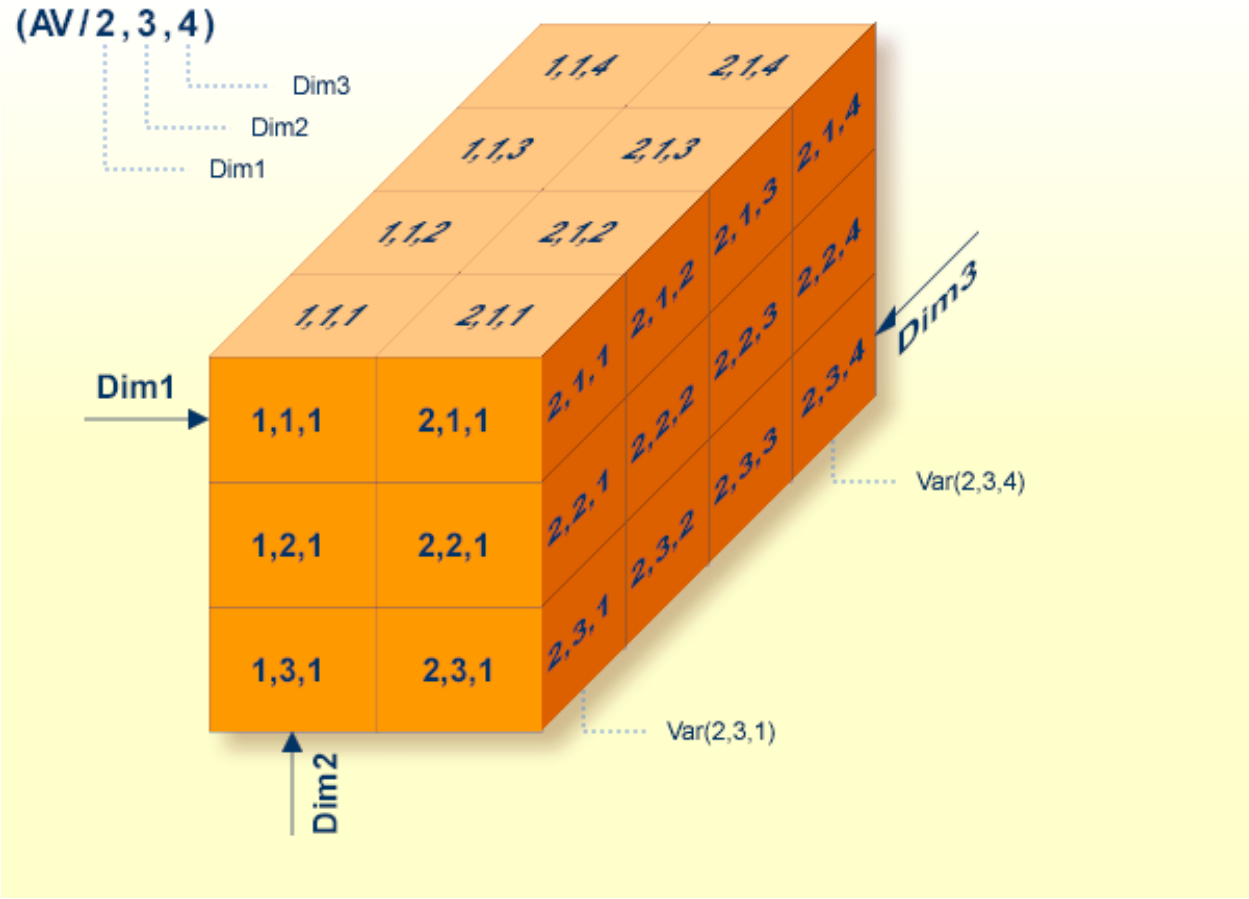
```
...
1 NAMES (A100/V) /* 1 dimensional array */
1 TUPLES (A100/V,V) /* 2 dimensional array */
1 TRIPLES (I1/1:V,1:V,1:V) /* 3 dimensional array */
...
```

Example of Arrays with Variable Upper-bounds and Maximum

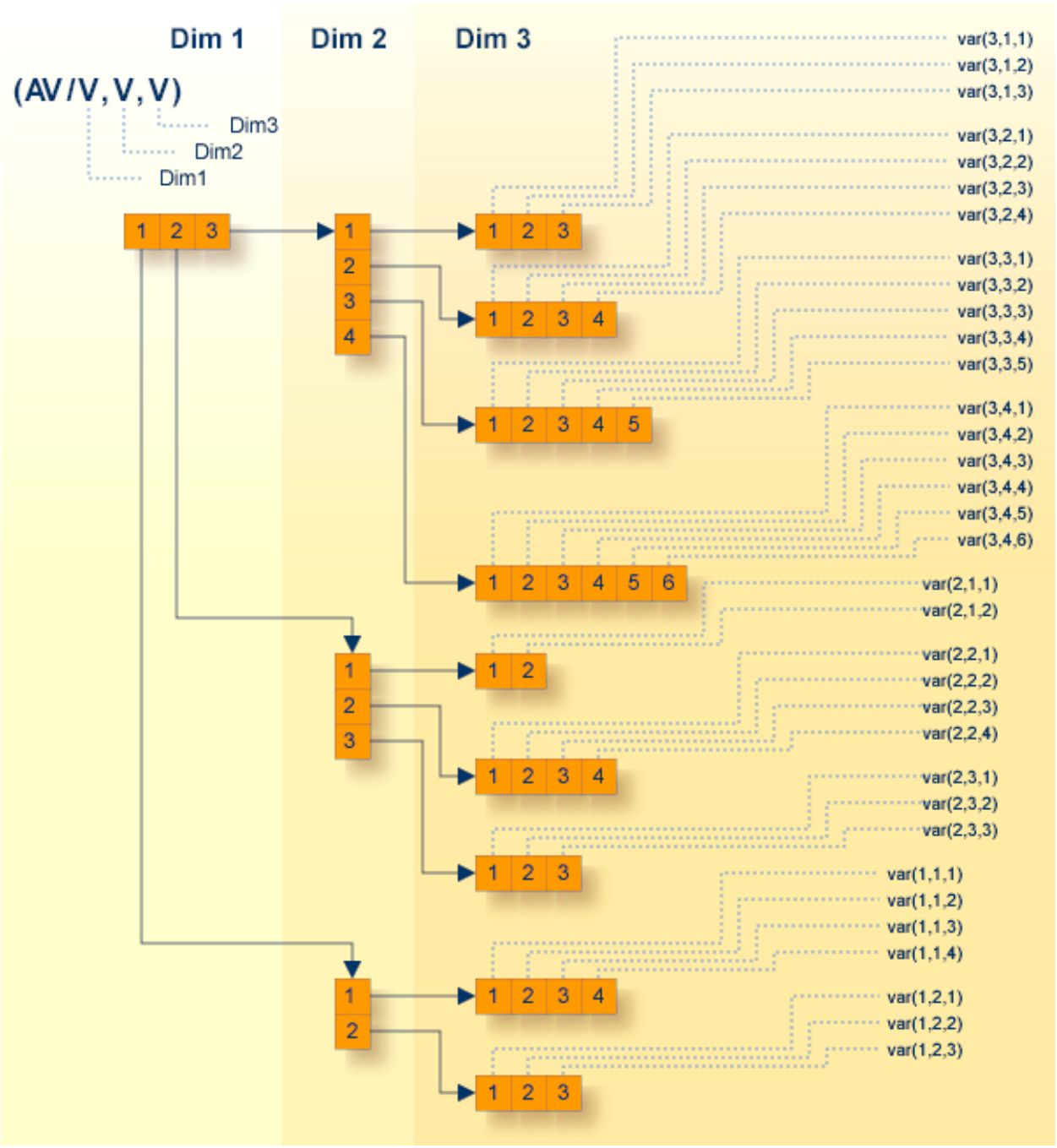
```
...
1 NAMES (A100/V10) /* 1 dimensional array */
1 TUPLES (A100/V10,V10) /* 2 dimensional array */
1 TRIPLES (I1/1:V20,1:V20,1:V20) /* 3 dimensional array */
...
```

Caution: Mixed arrays with fixed upper bounds and variable upper bounds are not supported.
`(I2/1:V,20,V)` is not permitted.
`(I2/V10,30)` is not permitted.
Mixed arrays with variable upper bounds with maximum and without maximum are not supported.
`(I2/V10,V20,V)` is not permitted.

Three-dimensional Array with Fixed Bounds



Three-dimensional Array with Variable Upper Bounds



In the illustration above, the vectors of the second dimension have different lengths. The first vector has a length of 4, the second a length of 3 and the third a length of 2. The same is true for the third dimension with vector length of (3,4,5,6) (2,4,3) and (4,3).

Please note this kind of an unbounded array is not possible if you are using COBOL as the endpoint. In COBOL, all vectors in a dimension have the same length. A 2-dimensional array forms a rectangle and a 3-dimensional array forms a cuboid.

- For the *COBOL Wrapper*, see *Mapping Fixed and Unbounded Arrays*.
- For the *IDL Extractor for COBOL*, see *Tables with Variable Size - `DEPENDING ON` Clause*.

attribute-list

Attributes describe further parameter properties to correctly map the parameter to the target platform or to optimize the parameter transfer.

Syntax

`attribute-list`	`::= [ aligned-attribute ] [ direction-attribute ] [ ims-attribute ]`
`aligned-attribute`	`::= ALIGNED`
`direction-attribute`	`::= IN | OUT | IN OUT | INOUT`
`ims-attribute`	`::= IMS`

Description

`aligned-attribute`	■ The `aligned` attribute (mainly) belongs to the server and describes a different alignment from the compiler's default of the server interface. ■ The aligned attribute is relevant for the programming languages COBOL and PL/I in the RPC server environments batch, CICS and IMS. ■ Programming languages other than COBOL and PL/I do not consider it in the interface of the generated servers. ■ RPC Clients send the aligned attribute within the RPC data stream to the RPC server, where it is considered (the related parameter is aligned) by the RPC server, if relevant. ■ See *Mapping the `aligned` Attribute* in the respective Wrapper documentation for information on whether your client environment supports sending aligned attributes.
`direction-attribute`	The direction attribute optimizes parameter transfer. ■ In data is passed from client to server. ■ Out data is passed from server to client. ■ In Out data is passed in both directions.

	■ The direction of group members is inherited from the parent group. Thus only the direction information of the top-level fields (level 1) is relevant. Group fields always inherit the specification from their parent. ■ A different specification given with the group members is ignored. ■ The direction of members of a `structure-definition` is inherited from the `structure-parameter-definition`. Thus only the direction information of the `structure-parameter-definition` (structure reference) is relevant. Structure fields always inherit the specification from their reference. ■ A different specification given with the structure members is ignored. ■ When no direction is specified, In Out is used as the default. ■ See *Mapping the Direction Attributes `IN`, `OUT`, `INOUT`* in the respective Wrapper documentation.
`ims-attribute`	The ims-attribute marks PCB (Program Communication Block) parameters for the target platform IMS (IBM's Information Management System). ■ The `ims-attributes` are considered when servers for the target platform IMS are generated with the COBOL Wrapper and the PL/I Wrapper. ■ The `ims-attributes` are obsolete for clients and other wrappers than COBOL and PL/I and are ignored. ■ The `ims-attribute` is only relevant on top-level fields (level 1). Group fields always inherit the specification from their parent, thus a different specification is ignored.

Example of aligned-attribute

```
1 PERSON_ID (NU12) ALIGNED
```

Example of direction-attribute

```
...
1 PERSON_ID (NU12) IN
1 PERSON_NAME (A100) OUT
...
```

Example of ims-attribute

```
...
1 PERSON_ID         (NU12)  IN OUT
1 PERSON_NAME       (A100)  IN OUT
1 DBPCB             IMS
   2  DBNAME           (A8)
   2  SEG-LEVEL-NO     (A2)
   2  DBSTATUS         (A2)
   2  FILLER           (A20)
...
```

5 The Software AG IDL Compiler

The Software AG IDL Compiler generates interface objects, skeletons and wrappers. It uses a Software AG IDL file and a template file that controls the generated output.

Introduction

The IDL Compiler is used to generate stubs, skeletons and wrappers from two specific input files:

- the IDL file (extension .idl), which describes the interface between client and server.
- the template file (extension .tpl), which controls the generated output files and their contents. In principle the template describes the target programming language.

The IDL Compiler first reads through the IDL file and builds tables and structures that form an internal representation of the interface. If there is a related client-side server mapping file, it is also read implicitly (see *CVM File*). The IDL Compiler then loops through this internal representation and uses the template file to generate its output, the source code for the target programming language.

The following wrappers use the IDL Compiler as their generation tool:

- *EntireX DCOM Wrapper*
- *EntireX C Wrapper*
- *EntireX .NET Wrapper*
- *EntireX COBOL Wrapper*
- *EntireX PL/I Wrapper*

The IDL Compiler and the template files for the target programming languages above are fully integrated in the *EntireX Workbench*. For automation purposes, the IDL Compiler can also be started at the command prompt.

Starting the IDL Compiler

To start the IDL Compiler

- At a command prompt, enter

```
Java -classpath "%ProgramFiles%\software
ag\entirex\classes\exxidlcompiler.jar;%ProgramFiles%\software
ag\entirex\classes\saglic.jar" "-Dsagcommon=%CommonProgramFiles%\Software AG"
com/softwareag/entirex/idlcompiler/TplParser -t template file [-Doption=value]
[-Fbasename] [-Ppreprocessor variable] [-ooutput-directory] [-Ttrace-level]
[-deprecated] -idl idlfile.idl
```

Parameter	Description
`-help`	Displays help using the command-line options.
`-D option=value`	Passes the option to the templates. See also ***Using Options***.
`-F basename`	Indicates that the output file base name follows. It is used as given - thus it should be provided without a path and extension. Only relevant if supported by the template used. Default is the base name of the *idl-file* without path and extension. See ***Specifiying the Name of the Output File***.
`-t template`	Name of the initial template file.
`-P preprocessor variable`	Specifies a preprocessor variable that can be controlled in template files with `#ifdef`, `#elif`, `#else`, and be closed by `#endif`. Only relevant if supported by the template used. See ***Using Template #if Preprocessing Statements***.
`-o output-directory`	Specifies the directory for the output file. See ***Specifiying the Name of the Output File***.
`-T trace-level`	Trace information is entered as comment lines into the generated output.

Trace level	Description
0	No trace output will be created. This is the default trace level.
1	Generates only template file name and the line number in the output file. This trace level helps to identify the specific line in the template file.
2	This is the full tracing mode for the template file. Every action of the IDL Compiler will be displayed.
3	Additional traces for preparation of the output statement will be created.

`-deprecated`	Deprecated mode allows compilation of templates for statements already deprecated. Each deprecated statement will create a warning.
`-idl idl-file`	Name of the IDL file. Multiple IDL files can be provided.

IDL Compiler Usage Examples

Calling the IDL Compiler under UNIX

The following applies to the UNIX Bourne, Korn and C shell.

```
$JAVA_HOME/java -Dsagcommon=/opt/softwareag/EntireX\common\conf
-Dentirex.home=$EXXDIR -classpath
$EXXDIR/classes/exxidlcompiler.jar:$EXXDIR/classes/saglic.jar
com/softwareag/entirex/idlcompiler/TplParser -t $EXXDIR/template/client.tpl -idl
aaclient.idl
```

An *erxidl.bsh* shell script file is provided with a preconfigured invocation of the IDL Compiler. In addition, the "-deprecated" mode is set as default in this shell script file.

```
erxidl.bsh -t ...\EntireX\template\client.tpl -idl aaclient.idl
```

Calling the IDL Compiler under Windows

```
java -classpath "%ProgramFiles%\software
ag\entirex\classes\exxidlcompiler.jar;%ProgramFiles%\software
ag\entirex\classes\saglic.jar" "-Dsagcommon=%CommonProgramFiles%\Software AG"
com/softwareag/entirex/idlcompiler/TplParser -t ...\EntireX\template\client.tpl
-idl aaclient.idl
```

An *erxidl.bat* batch file for the Windows command shell is provided with a preconfigured invocation of the IDL Compiler. In addition, the "-deprecated" mode is set as default in this batch file.

```
erxidl -t ...\EntireX\template\client.tpl -idl aaclient.idl
```

Writing your own Wrappers and Stubs

Additional programming languages can be adopted with user-written templates (see ***Writing Template Files for Software AG IDL Compiler***). The syntax for IDL template files in a formal notation is presented in the section ***Grammar for IDL Template Files***.

Integration with the IDL Editor can be accomplished using the plug-in technique provided (see *Using EntireX Custom Wrappers*).

6 Writing Template Files for Software AG IDL Compiler

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.

Caution: 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 provides an introduction on how to write template files. The syntax for IDL Template Files in a formal notation is presented in the document ***Grammar for IDL Template Files***.

Coding Tempate Files

It is the combination of control and output statements (see *control_statement*, *output_statement* and ***Using Output Statements in the Template File***) that provides the full definition of the target programming-language source code.

Usually a template file has *definition-statement* grouped together at the beginning; these are followed by loop_statements:

```
; type definitions
%using A "char %name%index"
....
; loop libraries
%library
{
          ....
          ; loop programs
          %program
          {
                   ....
                   ; loop parameters
                   %name
                   {
                            ....
                   }
           }
}
```

Using Output Statements in the Template File

Output statements (see *output_statement*) provide the actual templates of the target-language source code. Output statements are text strings enclosed in double quotes.

These text strings may contain *output_substitution_sequence*, *output_formatting_sequence*, *output_escape_sequence* and *output_of_variable*.

Substitution Sequences

Substitution sequences are identified by a preceding % and are substituted by their actual contents during generation.

Example: `"This is a text string. The library name is %library. \n"`

The IDL Compiler provides substitution sequences for the current library name, program name, parameter name, type, etc. Some substitution sequences can only be accessed in their corresponding loop statement. For example a `%program` substitution sequence is only valid within an active program loop (see *loop_over_programs*). See *output_substitution_sequence* for a list of valid substitution sequences and description.

Formatting Sequences

Formatting sequences are identified by a preceding \

Example: `"\n is a Formatting sequence"`

See *output_formatting_sequence* for a list of valid formatting sequences

Escape Sequences

Escape sequences are identified by a preceding \\

The escape character is used to change the meaning of special characters (&, ? and # etc.) back to their normal meaning. Special characters are used to access variables (see *output_of_variable*).

Example:`"\\&"`.

See also *output_escape_sequence* and ***Using Verbatim Mode***.

Variables

The output of variables is forced when the special characters &, ? or # occur before the variable name (see *variable_name*) in output statements (see ***Using Output Statements in the Template File***)

Example: `"?A is the output of a variable"`

See also *output_of_variable*

Generating Programming-language-specific Type Definitions

The substitution sequence `%type` is usually used in a parameter loop (loop_over_parameters) to generate programming-language-specific type definitions. Before the parameter loop all IDL data types (with *definition-of-base-type-template* statements) and the dimension information (with *definition-of-index-template* statements) must be specified.

Example

IDL data type I2 can be specified as follows in a C program:

```
%using %index "" "[%1_index]" "[%1_index][%2_index]"
"[%1_index][%2_index][%3_index]"
%using I2 "short %name%index;"
```

`%using %index` is the control_statement for the dimension information (*definition-of-index-template*). How the following strings are used depends on the dimension of the parameter. The first empty string is used for scalar parameters, the second for 1-dimensional parameters, the third for 2-dimensional parameters and the fourth for 3-dimensional parameters.

`%using I2` is the control_statement for the IDL data type I2 (see *definition-of-base-type-template*), "short %name%index" is the output_statement for this data type. `%name` and `%index` are substitution sequences. `%name` will be replaced by the variable name and `%index` will be replaced by any dimension information.

If an input IDL file contained the following parameter definitions:

```
1 Field-1   (I2)
1 Field-2   (I2/1:8)
1 Field-3   (I2/1:4,4:7)
```

then, based on the above template specifications, all references to the `%type` substitution sequences in any `output_statement` would be replaced by

```
short Field_1;
short Field_2[8];
short Field_3[4][4];
```

Generating Programming-language-specific Names

Special characters within some substitution sequences e.g. `%library`, `%program` and `%name` can be changed during generation to provide valid names for the target programming language. The IDL Compiler supports generation of names for the programming languages C, C# and COBOL (see *output_control_lower_upper* and *output_control_sanitize*).

Target Programming Language	Class Names	Function Names	Variable or Parameter Names
C	not applicable	%UpperCase- %LowerCase+ %Sanitize+	%UpperCase- %LowerCase+ %Sanitize+
C#	%UpperCase- %LowerCase- %SanitizePascalCased+		%UpperCase- %LowerCase- %SanitizeCamelCased+
COBOL	not applicable		%SanitizeCobol+ %UpperCase+ %LowerCase-

The default programming language when you do not code any `output_control_lower_upper` and `output_control_sanitize` statements in your template is C.

Inserting Comments in the Template File

Comments

- are identified by a ";" in a line and
- are terminated by the end of line.

For example:

```
; This is a comment
                ; So is this.
"output text followed by a comment" ; here is the comment
```

Whereas this is an output statement:

```
"an output text with a semicolon ;"
```

Using Verbatim Mode

If your output is going to contain many special characters, you may enter verbatim mode. Then all characters are written to the output as typed. The only sequences recognized in this mode are the escape sequences (see *output_escape_sequence*).

To enter verbatim mode

- Use the command `%verbose+` (see *output_control_verbose*).

  Example: In verbatim mode, you enter "`&`" to insert an ampersand.

Using Options

The IDL Compiler supports options within templates.

You can pass them with the parameter `-D` to the IDL Compiler (see ***Starting the IDL Compiler***).

Options

- can be used in output statements (see *output_of_variable*)
- can be used in logical condition criteria (see *compare_strings*) in %if (see *if_statement*) and %while (see *loop_of_while*) statements
- are case-sensitive, i.e. hugo and HUGO are distinct options

Specifiying the Name of the Output File

The name of the output file is controlled by the %file statement.

If the `%Format` substitution sequence in a file (`%file`) statement is used, the base name can be provided with the IDL Compiler parameter `-F` (see ***Starting the IDL Compiler***).

If no base name is provided with the `-F` IDL Compiler parameter, the base name of the IDL file without path and extension is used as the default of the substitution sequence `%Format`.

See the following excerpt from a template file:

```
%file "C%F.c"
```

When the IDL Compiler is called with

- `erxidl -t client.tpl .. \MyDirectory\example.idl`

  an output file with the default base name *Cexample.c* of the IDL file is created
- `erxidl -t client.tpl -Ftest example.idl`

  an output file with the name *Ctest.c* is created

See also the IDL Compiler option `-o` (see ***Starting the IDL Compiler***) on how to specify the directory for the output file.

Redirecting the Output to Standard Out

The output can be redirected to standard out with an environment variable (see ***Using Options***), e.g. `NOOPEN`. This is optional.

See the following excerpt from a template file:

```
%if "$(NOOPEN)" <> "1" %file "C%library.c" ;
```

When the IDL Compiler is called with

- `erxidl -t client.tpl -D NOOPEN=1 example.idl`

  the output is redirected to standard out
- `erxidl -t client.tpl example.idl`

  the output is directed to the file *Cexample.c* as specified in the template.

Using Template #if Preprocessing Statements

The IDL Compiler supports `#ifdef`, `#elif`, `#else` and `#endif` preprocessing statements similar to the C compiler preprocessor.

You can use preprocessor variables with the option -D (see *Starting the IDL Compiler*).

Additional rules for `#if` preprocessing statements are:

- If `#elif` is used, it must follow `#ifdef`.
- If `#else` is used it must follow either `#ifdef` of `#elif`.
- `#endif` must always close the `#ifdef` statement.
- Embedded preprocessor statements or logical concatenation of definitions are not allowed.

See the following excerpt from a template file:

```
#ifdef Definition_1
               "/* codes of -PDefinition_1 */\n"
               %name
               {
               :
               }
       #elif Definition_2
               "/* codes of -PDefinition_2 */\n"
               %name
               {
               :
               }
       #else
               "/* codes of neither Definition_1 nor Definition_2 */\n"
               %name
               {
               :
               }
       #endif
```

When the IDL Compiler is called with

- `erxidl -t template_file -PDefinition_1`

  the template statements "`/* codes of -PDefinition_1 */\n`" between the `#ifdef` and first `#elif` statement are interpreted.

- `erxidl -t template_file -PDefinition_2`

  the template statements "`/* codes of -PDefinition_2 */\n`" between the first `#elif` and second `#elif` statement are interpreted.

- `erxidl -t template_file`

  the template statements "`/* codes of neither Definition_1 nor Definition_2 */\n`" between the `#else` and `#endif` statement are interpreted.

See the following preprocessing statements with invalid syntax:

```
..
#ifdef (MY_VERSION) ; brackets are not allowed
#endif
#ifdef MY_VERSION || HIS_VERSION ; logical OR is not allowed
..
#endif
#ifdef (MY_VERSION)
..
#ifdef (MY_NEW_VERSION) ; embedded #ifdef is not allowed ↩
```

Using Template #include Preprocessing Statements

The IDL Compiler supports `#include` preprocessing statements similar to the C compiler preprocessor. All statements in the included template file are simply embedded.

To find included template files, use the IDL Compiler option `-I` and add a list of directories that form a search path (see ***Starting the IDL Compiler***).

- First the IDL Compiler searches for templates in the directory of the initial template.
- When no template is found in the directory of the initial template, all directories specified with `-I` are searched in the order of occurrence

Additional rules for `#include` preprocessing statements are:

- A maximum of 32 templates can be included in a generation process.
- An included template file can include further template files.
- Recursive inclusion of template files is not permitted.
- All variables can be accessed in all included template files as well as in the starting (root) template.

The compiler searches for included templates.

See the following excerpt from a template file:

```
#include "template.tpl"
```

Using Template #trace Statement

The IDL Compiler supports the `#trace`*`tracelevel`* statement to enable and disable template tracing within a certain block of the template. The usage of *`tracelevel`* is the same as the command-line option "`-T`".

See also compiler option "`-T`" under ***Starting the IDL Compiler*** for trace level values.

Example

```
#trace 2   ; enable tracing on trace level 2
%compute i "0"
%while "&i" < "10"
{
  %compute i "&i + 1"
}
#trace 0   ; disable
```

7 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.

Caution: 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 futher 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 stetement 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 ouput 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 piority 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*.
%eLength	**Data Type** / **Description** A,K,B,I,F,U — Logical length as set in the IDL file. AV,KV,BV,UV — Maximum logical length as set in the IDL file. L — Type L has no explicit logical length in the IDL file. However, the length is always set to 1. N,NU,P,PU — 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. T — Type T has no explicit logical length in the IDL file. However, the length is always set to 12.

Sequence	Meaning	
	Data Type	**Description**
	D	Type D has no explicit logical length in the IDL file. However, the length is always set to 6.
`%file`	Inserts the current filename into the output.	
`%Format`	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.	
`%index`	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*.	
`%LibCount`	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`.	
`%library`	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*.	
`%member`	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*.	
`%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*.	

Sequence	Meaning
%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.
%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

Sequence	Meaning
	substitution sequence). This substitution sequence can only be used in an active *loop_over_parameters*.
%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 client-side server mapping file (see *CVM 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 CVM file, an empty string is inserted
%SVMFormatArea	Inserts the format area contained in a related CVM 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 CVM file, an empty string is inserted
%SVMValueArea	Inserts the value area contained in a related CVM 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 CVM file, an empty string is inserted
%SVMStringArea	Inserts the string area contained in a related CVM 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 CVM file, an empty string is inserted
%SVMRpcProtocol	Inserts the RPC protocol version contained in a related CVM 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 CVM 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_list` 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_statement` 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
```