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*. This chapter covers the following topics:

- Meta Definitions
- Syntax of the IDL File
- library-definition
- program-definition
- structure-definition
- parameter-data-definition
- simple-parameter-definition
- group-parameter-definition
- structure-parameter-definition (IDL)
- array-definition
- attribute-list

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* to C | COBOL | DCOM | .NET | Java | Natural | PL/I. • 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* under z/OS (CICS, Batch, IMS) | UNIX | Windows | Micro Focus | BS2000/OSD | z/VSE (CICS, Batch) | IBM i. • 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* to C | COBOL | DCOM | .NET | Java | Natural | PL/I. • 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* to C | COBOL | DCOM | .NET | Java | Natural | PL/I. • 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* under z/OS (CICS, Batch, IMS) | UNIX | Windows | Micro Focus | BS2000/OSD | z/VSE (CICS, Batch) | IBM i. • 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* to C | COBOL | DCOM | .NET | Java | Natural | PL/I. • 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* to C | COBOL | DCOM | .NET | Java | Natural | PL/I | RPG.
`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* to C | COBOL | DCOM | .NET | Java | Natural | PL/I. • 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* to C | COBOL | DCOM | .NET | Java | Natural | PL/I | RPG.

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* to C | COBOL | DCOM | .NET | Java | Natural | PL/I | RPG. • 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* to C | COBOL | DCOM | .NET | Java | Natural | PL/I | RPG.
`unbounded-array`	• Unbounded arrays are not supported by all endpoints, or are supported with restrictions. • See *Mapping Fixed and Unbounded Arrays* to C | COBOL | DCOM | .NET | Java | Natural | PL/I | RPG.
`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 an `array-index` is of the format `[1:]V[maximum-upper-bound]` it describes an unbounded array with variable bounds. ○ If `maximum-upper-bound` is given, the variable number of occurrences (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 occurrences 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.
`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 */
...
```

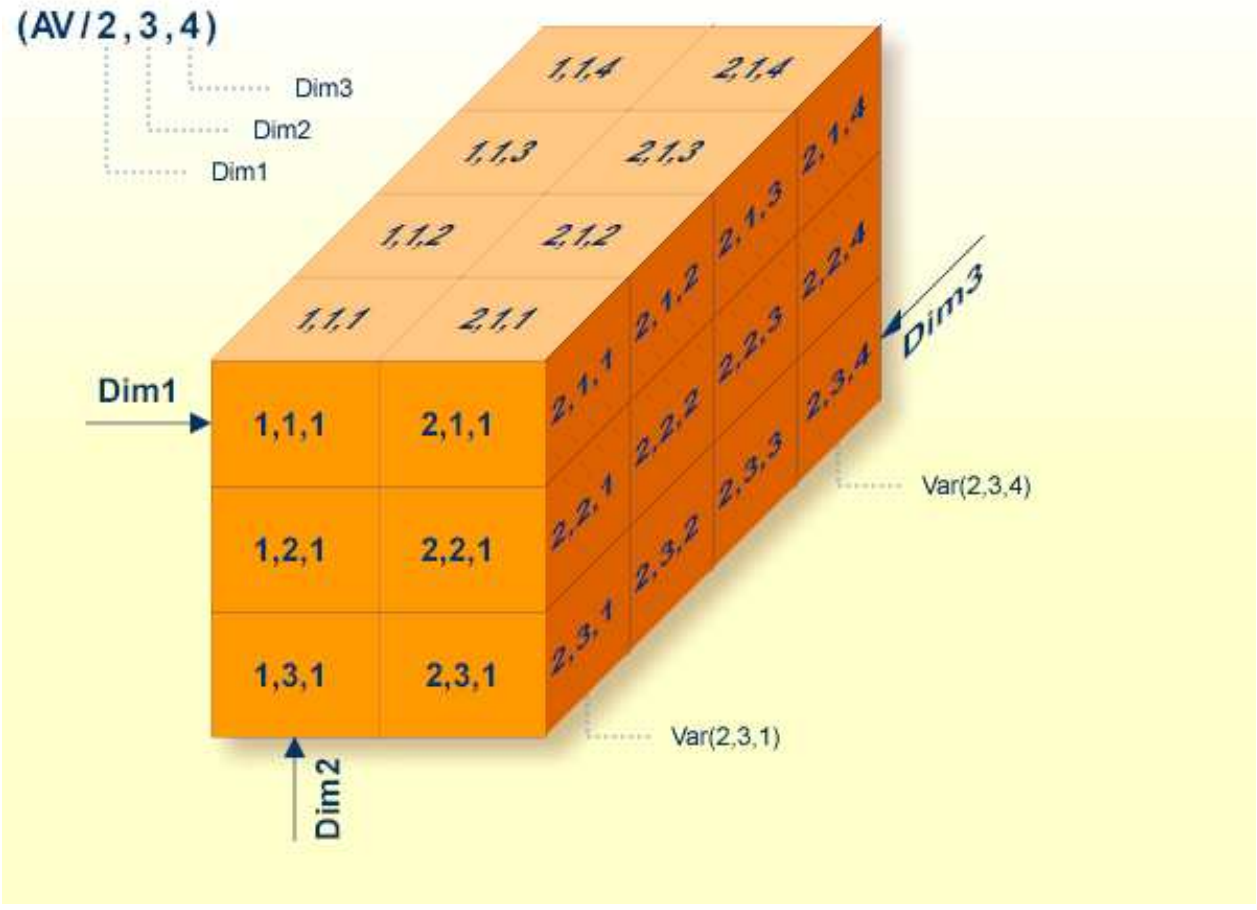
Example of Arrays with Variable Upper-bounds and Maximum

```
...
```

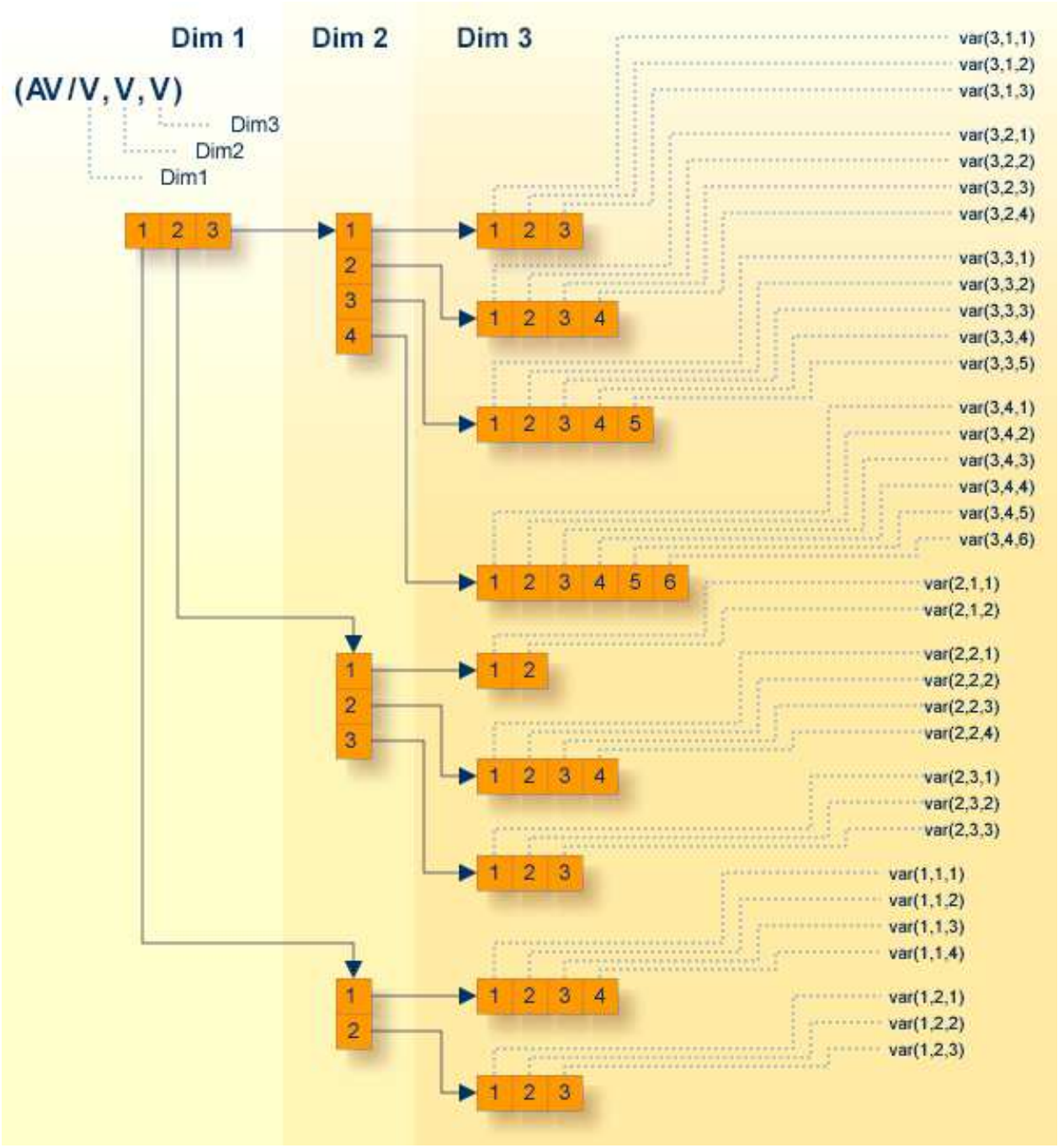
```
1 NAMES (A100/V10) /* 1 dimensional array */
1 TUPLES (A100/V10,V10) /* 2 dimensional array */
1 TRIPLES (I1/1:V20,1:V20,1:V20) /* 3 dimensional array */
...
```

Warning:
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* to C | COBOL | DCOM | .NET | Java | Natural | PL/I 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`* to C | CL | COBOL | DCOM | .NET | Java | Natural | PL/I | RPG.
`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)
...
```