CodeMagus

objtypes: Configuring for Object Recognition, Generation and Manipulation

CML00018-01

Code Magus Limited (England reg. no. 4024745)
Number 6, 69 Woodstock Road
Oxford, OX2 6EY, United Kingdom
www.codemagus.com

CodeMagus

December 15, 2020

1 Introduction

There are a number of cases where the automatic determination of an object is required outside of the context where that object would normally reside, or where an object of a particular format or type requires manipulation in a context where the details of the type of that object need to be taken into account. The interpretation, manipulation or generation of records of files and messages of network traffic are just two examples of such cases. In such situations the type of the object or the collection of the type of the objects is manifest by the context in which they are processed. An example of this might be a program designed to process a set of records as part of a specific file type or to process network messages as part of a specific service. In these cases, it is not uncommon for the program to determine from a number of options which of a small set of types the record or message belongs. That the message belongs to that small set is is assumed. To use an example, a program used to process records or messages of types A, B or C, would check the whether some predicate, usually over a single record-type or message-type field, indicates whether the record or message is of type A, B or C. Typically, such programs reject records or messages which appear not to be of type A, B or C. Where a record appears to be of type A, B or C the program would typically accept that the record is of type A, B or C and, at best and usually, report that the record or message of type A, B or C is ill-formed if the record or message really is not of the asserted type.

The `objtypes` artefacts are designed to lift the information out of the context of the record or message processing programs where types of messages or records can be determined in a context independent of the processing program of those records or messages. This still restricts the type of a message or record to one of the types acceptable by the containing type. For this reason the records or messages are abstractly referred to as objects of a particular type known within a collection of object types. Hence the name of the library and the artefact type.

It is intended that the collection of types is exhaustive of the types found in a particular collection type (type of file or type of network service or type of transaction) and hence are all reflected in the same configuration `objtypes` artefact. The artefact is designed to be both machine and human readable and to provide both configuration data for a tool processing collections of the specified types.

An `objtypes` collection of types is a mapping of an object known to be one of a collection of types (i.e. known to be one of the records of a file or known to be one of the messages of a network service) to its exact type. Because the type in turn maps to the actual meta-data of the type, the mapping is a mapping down to the meta-data of the object. The means by which this mapping is determined is by the evaluation of a predicate associated with the type and evaluated over the object itself.

The description of the meta-data is outside of the scope of this artefact. This is deliberate and desirable. Typically, the meta-data exists for a purpose other than the `objtypes` artefact. For example, a copybook for a file, or a schema for a database entity. These meta-data artefacts are maintained for the application systems that use them and the meta-data is not necessarily owned by the owner of the `objtypes` artefacts. The

tools, for example, that use the objtypes are used not necessarily only used in the development process. For this reason the objtypes artefacts only refer to the meta-data and they do not redundantly replicate the meta-data.

There are a number of desirable side effects of using the objtypes artefact approach for configuration:

- The first is that the actual detail meta-data is referenced and not replicated. This allows the meta-data to remain with the owners and to be changed by them without too much regard for the validity of the objtypes artefacts in general. This ensures that the cost of maintaining an objtypes artefact is kept to a minimum.
- Because the objtypes artefacts describe their mappings independently of their actual use, they are generically usable and a particular objtypes artefact often appears in the configuration of different tools. This in turn both amortises the cost of maintenance of the artefact and puts pressure on the requirement for the maintenance of the validity of an objtypes artefact.
- This document describes the objtypes configuration. There is also a mapping between a containers name and the objtypes artefact that describes the types of the objects within the container. Because the objects that we deal with in this manner are always sequential, the configuration of that mapping is referred to as seqtypes are is described elsewhere.
- The component that processes an objtypes artefact is a common component which runs unchanged on various platforms and which is independent of the tool that processes the resultant data structure created by the component by processing a objtypes artefact. This, and the point above, makes sure that the investment in the high-degree of engineering of the component has a continuing return on investment as do any tools that use the component.

The application interface is described in the objtypes library [1].

2 objtypes Configuration

An objtypes artefact is a textual description of the mapping of an object to the type and meta-data that describes that object within a collection of possible types. This description follows a formal grammar whose syntax and semantics will be described in this section.

2.1 Lexical Elements

The lexical elements used to describe the objtypes grammar are designed to make the objtypes artefacts reasonable and their semantics manifest. However, some of the

elements referred to are defined as part of the application related meta-data which are referred by the `objtypes` artefacts and the lexical elements of those artefacts as they are used in the language processing systems are not quite as convenient as they could be. The primary example (currently the only one implemented) is the COBOL copybook in which the element names can have a minus sign or dash in them. This conflicts with the more general 'denser' expressions that have become customary in third generation programming languages after COBOL. For this reason, the dash is always interpreted as an under-score character.

At the lexical element level white-space characters such as the space character, new-line characters, carriage returns and tab characters are ignored and their presence or otherwise between other lexical elements does not change the meaning of `objtypes` artefact.

Similarly, the presence or absence of comments does not change the meaning of an `objtypes` artefact. Comments are introduced anywhere in the text between lexical elements using two adjacent dash characters and the comment extends from the double dash to the next new-line character:

```
-- Test object type specification processed by testprog.c.
-- This collection of types is required by the testprog.c
-- program to test the type collection library objtypes.
```

Other lexical elements include the detection of literals, identifiers, reserved words and non-word operators that are formed with sequences of characters.

2.1.1 Literal Constants

There are a number of types of literals:

- **Strings:** A string literal is introduced by a quotation mark character or the apostrophe, which is not considered part of the value of the literal, a sequence of characters which excludes the opening quotation mark or apostrophe or newline characters and extends until the next matching quotation mark character or apostrophe, which again is not considered part of the value of the literal. The collating sequence of assumed for the representation of string literals is determined by the default collating sequence of the machine or the explicitly stated collating sequence specified for the data.
  - `'abc'` and `"abc"` are string literals with the same value comprising the lower case character sequence `abc`.
  - `'"NO!" exclaimed'` and `"it's usually possible"` are string literals in which the non-delimiting apostrophe or quotation mark is required in the string.

- **Hexadecimal strings:** Some representations are required to be concrete representations independent of any assumed collating sequence or byte ordering. For this purpose, hexadecimal literals are available. A hexadecimal literal is introduced using the character 'x' or 'X' followed by an apostrophe or quotation mark character, followed by a sequence of the digits 0 ... 9 and the characters 'a', 'b','c', 'd', 'e', 'f' 'A', 'B', 'C', 'D', 'E' and 'F'. The sequence is terminated by a matching apostrophe or quotation mark character. The characters 'x', 'X', the quotation marks or the apostrophe characters are not part of the literals value and there must be an even number of hexadecimal digits between the quotation marks and the apostrophe characters.
- **Numbers:** Numeric literals are formed by a sequence of the decimal digits 0 ... 9 forming the integer portion of the number. This sequence can be preceded by an optional '+' or '-' sign indicating the sign of the number (a number without a sign is assigned to be non-negative). The number may be followed by an optional decimal fraction introduced with a decimal point '.' followed by a sequence of decimal digits.

  The following are valid numbers: `100`, `100.000`, `+100`, `-100.123`, `0100.123`.

2.1.2 Reserved Words

There are number of reserved words which are used as key-words in the grammar. These words cannot be used in ordinary identifiers. Reserved words are not case-sensitive. Reserved words are used to introduce the syntactical constructs described in Section 2.2 on page 5.

`path`	`options`	`type`	`title`
`book`	`map`	`if`	`include`
`exclude`	`base`	`bias`	`when`
`div`	`mod`	`and`	`or`
`not`	`like`	`ascii`	`ebcdic`
`endian_big`	`endian_little`	`verbose`	
`bind_wsm`	`bind_bsm`	`omit_fillers`	
`lazy`	`attributes`	`suppressed`	

2.1.3 Composite Operators

Single character, non-word operators have their usual meaning and it is not necessary to distinguish them from other single characters not forming part of other lexical elements. There are a few composite operators which form lexical elements different from the sequence of characters that form them. These are:

Element	Meaning
<>	Relational operator for not-equal
>=	Relational operator for greater-than or equal
<=	Relational operator for less-than or equal

2.1.4 Identifiers

An identifier is a word comprising a sequence of characters starting with letter of the alphabet, and then followed by a sequence of letters of the alphabet, the decimal digits and the under-score character. An identifier can also start with a sequence of decimal digits provided this sequence is followed by a at least one letter of the alphabet or the under-score character. Letters of the alphabet may be either lower-case or upper-case. An identifier cannot have the spelling of a reserved word as the meaning of such lexical elements will have the meaning of the reserved word.

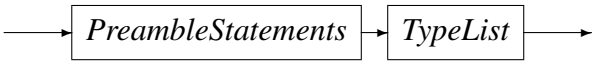
The following are valid identifiers: `control_data`, `00110_control_data`, `CONTROL_0001`, `CONTROL`.

2.1.5 Other Characters

All other characters are make up lexical elements in comprising the a single character and having the value of that character.

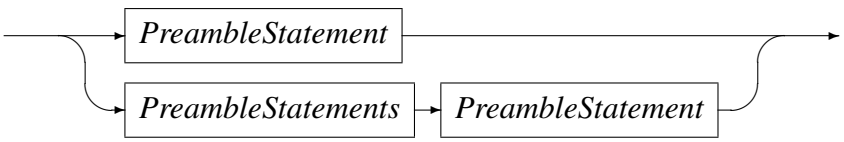
2.2 Syntax and Semantics

TypeConfig

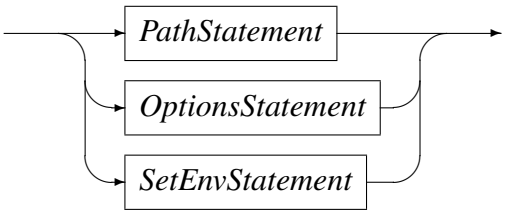


An `objtypes` configuration artefact or *TypeConfig* is a text file which comprises two sections: The first section *PreambleStatements* introduces global options and attributes which might not be inferred from the meta-data artefacts which are referenced in the type definitions. The second part of the *TypeConfig*, *TypeList*, details the individual types defined in the `objtypes` configuration.

PreambleStatements

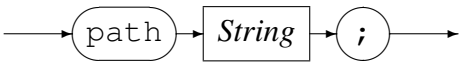


PreambleStatement



There are two types of *PreambleStatement* which may appear in any order and which may appear any number of times.

PathStatement



The *PathStatement* defines a path string which is used as a mask in order to find artefacts such as copybooks which might be referred to the *TypeList* section of the *TypeConfig*.

In the following example, a COBOL copybook named `testbook` referred to in the *TypeList* section of the *TypeConfig* will be assumed to be have the following full path and name:

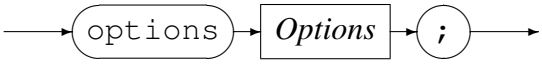
Given the following path statement:

```
path "/home/stephen/objtypes/%s.cbb";
```

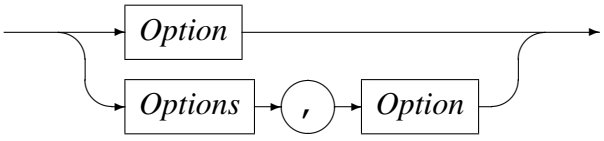
a COBOL copybook named `testbook` referred to in the *TypeList* section of the *TypeConfig* will be assumed to be have the following full path and name:

```
/home/stephen/objtypes/testbook.cbb
```

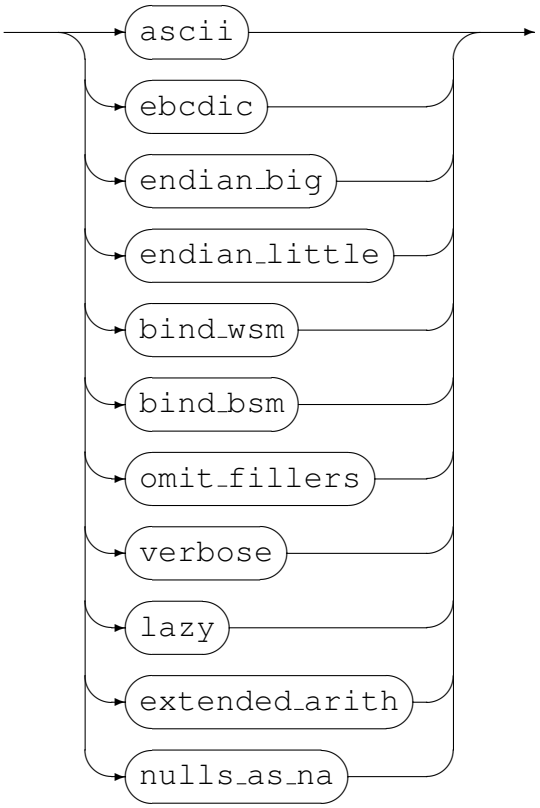
OptionsStatement



Options



Option



The *OptionsStatement* is used to supply meta-data attributes associated with the data which cannot be deduced from the meta-data (for in example from a COBOL copybook), or which the local machine cannot be used as the default value for such attributes. Some of these options are also used to determine the manner in which the `objtypes` library component is to process the data.

- The `ascii` and `ebcdic` attributes indicate that the underlying character set of the data presented to the `objtypes` library component for which one of the types in the *TypeList* may apply is based on the ISO-8 character set with code page 00819 (ISO/ANSI Multilingual) graphics (`ascii` option), or EBCDIC character set with code page 01047 (Latin 1/Open Systems) graphics (`ebcdic` option).

  These options should not be used together and if neither option is present then the architecture of the host machine will be used to provide a default value.

- The `endian_big` and `endian_little` attributes indicate that wherever binary data is present the byte ordering of the data follows the big-endian convention in which the most significant byte is found in the lowest byte address and that the bytes of successively lower significance are found in the following bytes in increasing byte address order (`endian_big`), or that binary date byte ordering follows the little-endian convention in which the most significant byte is found in the highest byte address and that the bytes of successively lower significance are

found in the preceding bytes in decreasing byte address order (`endian_little` option).

These options should not be used together and if neither option is present then the architecture of the host machine will be used to provide a default value.

- The `bind_wsm` and `bind_bsm` attributes describe the manner in which binary data is assigned to storage locations. The number of bytes required to store the values of a binary data item will determine the number of bytes assigned to the data item. These required and actual number of bytes are not always the same and are a function of whether or not the binary item is signed and whether not the machine intended to host the data is word-based or byte-based in its assignment of binary items to storage locations. Most machines are word based in which the number of bytes of a binary item is always a power of two with the minimum number of bytes being two, on these machines the assignment of the binding is said to follow the word storage mode binding (or WSM). This is default, but can be stated explicitly using the `bind_wsm` option.

  On some machines the assignment of binary data to storage follows the byte storage mode (or BSM) binding. In this mode the minimum number of bytes are used, but at least one, which will satisfy the sign and precision requested. This binding is selected with the `bind_bsm` option.

- Option `omit_fillers` is used when the processing of any COBOL filler items are to be excluded from processing. This option automatically sets the `hidden` attribute of any filler items [2].

- The `verbose` option is used to print as much diagnostic and progress information as possible and is useful in finding configuration errors.

- The `lazy` option is used to delay the loading of symbol tables by the underlying symbols library [2].

- The `extended_arith` option is used to relax the original COBOL restriction on the number of digits in a decimal number from 18 digits to 51 digits. This option support modern compilers where the number of significant digits supported is significantly expanded.

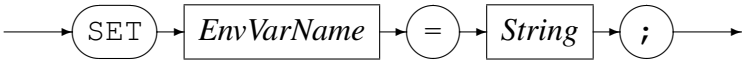
- The `nulls_as_na` option is used to format fields which contain all nulls as the string NA. This option can also be set by the the corresponding flags value in the application calling objtypes.

This following example illustrates the specification of a little-endian interpretation of data using the ISO-8 character set. This is an option which might be expected on Intel based machines:

```
options endian_little, ascii;
```

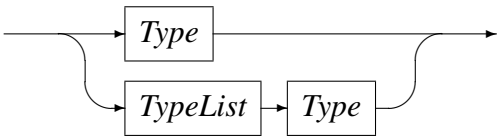
As with COBOL, copybooks may be included from within other copybooks. In order to to allow this without the amending the grammar of the imbedded copy-statements, the location of the copy libraries is supplied by environment variables of the same name as that of the copy-library name used in the copybook member or the default name SYSLIB. To support this, and for any other reason that environment variables may need to be set, preamble `SET` statements may be used.

SetEnvStatement



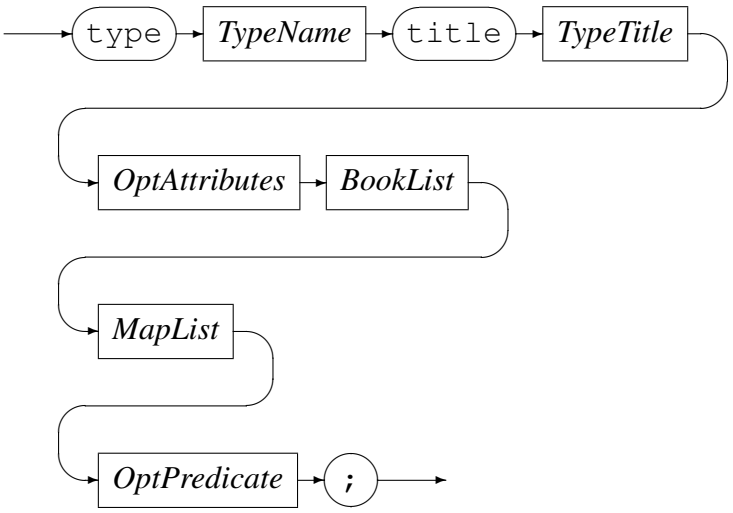
The *SetEnvStatement* causes the environment variable *EnvVarName* to be unconditionally set to the resolved value of the *String* at the point it is encountered in the preamble of the object types files.

TypeList



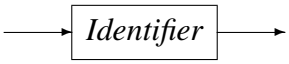
The remainder of the *TypeConfig* configuration comprises a list of definitions of each of the types that comprise the collection types.

Type



There are a number of elements which the comprise a single *Type*. These elements are for identification (*TypeName* and *TypeTitle*); for supplying type specific attributes to the *Type* (*OptAttributes*); for listing the meta-data elements used bye the *Type* (*BookList*); for specifying which pieces of which elements are present in the object of the defined *Type* and for specifying any position requirements for these elements; and specifying an optional predicate which indicates under what conditions a particular object is determined to be of the *Type* being defined.

TypeName



The format of the *TypeName* is that of an *Identifier* and is more suitable for a programming language or machine internal symbol reference to the *Type*.

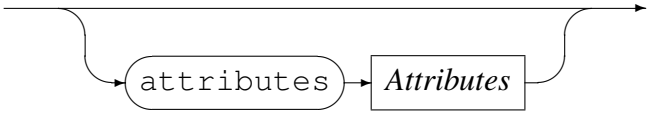
TypeTitle

The *TypeTitle* is more suitable for human consumption and is intended to be a short description of the type. This is not intended for a program to reference the type, but for a program to qualify a particular type when producing human-readable output.

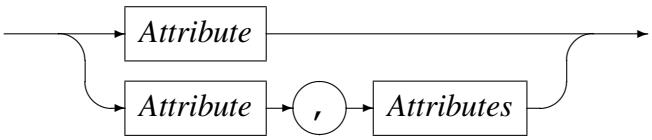
The following example shows the start of a *Type* definition in which the internal symbol and external title names are illustrated:

```
type types_a
   title "First sample type member --- record type A"
```

OptAttributes



Attributes

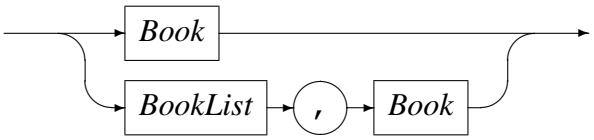


The attributes are optional and are indicate by a list of attribute keywords when specified. The list is initiated with the `attributes` keyword.

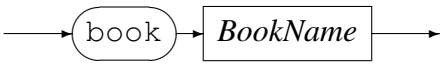
Attribute

The suppressed attribute is a way to communicate to certain programs that the *Type* is not to be processed, or that some aspects of the type are not to be processed. For example, this attribute may indicate to a program using the `objtypes` library that objects of this type are to be omitted from formatting; or that they are assumed to be missing from the input sequence.

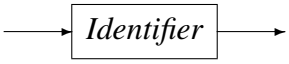
BookList



Book

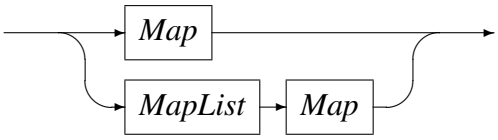


BookName

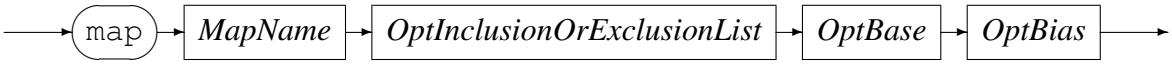


Each type must refer to the external meta-data elements that will be used to describe the mapping of the type. At present the meta-data artefacts being referred to are COBOL copybooks as indicated by the `book` keyword. The string representing the *Identifier* is used to form a full name of the artefact by editing the *Identifier* into the path string option. The completion of the preparation of the *Type* includes loading the symbols tables from these referenced artefacts. These symbols are referred to in the remaining syntactical elements of the *Type*.

MapList



Map

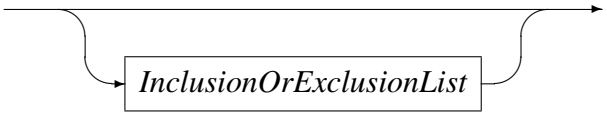


It is the *Map* constructs which pick up a map elements from the meta-data and assigns it to a portions of the buffer. The elements must be assigned in a manner which describes all meaningful portions of an object. In this context and map in the meta-data corresponds, for COBOL copybooks, the 01-level items present in the listed copybooks. This provides the means of resolving any ambiguity that might be present in the copybook because of the presence of multiple 01-levels.

MapName

The *MapName* is an *Identifier* which must be the name of an 01-level item loaded from one of the listed copybooks.

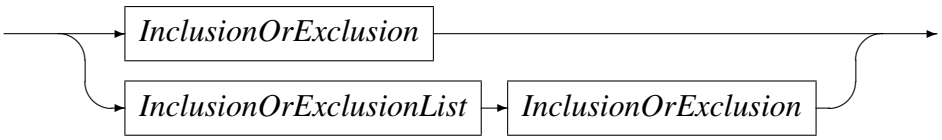
OptInclusionOrExclusionList



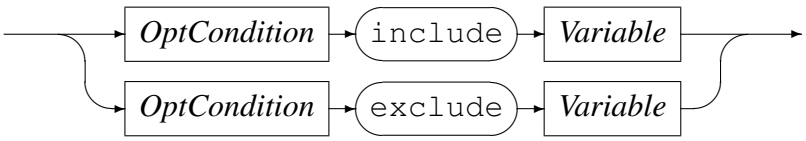
Apart from the possible of there being some ambiguity in the choice of structures (or 01-levels) in a meta-data artefact (such as a copybook), there is also the possibility that additional ambiguity might be present amongst the elements contained within that structure. The *InclusionOrExclusionList* is the means for resolving such internal ambiguity.

If no *InclusionOrExclusionList* constructs are present in a map, then the entire map will be included in the definition of the *Type*.

InclusionOrExclusionList



InclusionOrExclusion

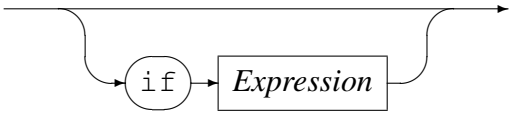


Typically, and specifically in the case of COBOL, PL/I, and C/C++, there is a hierarchical tree-like structure of meta-data elements each being identified by a symbol. This parent-child structure allows the recursive grouping of elements within a map with the leaf nodes having certain elementary types belonging to exactly one parent aggregate element. Any element within a map, whether elementary or aggregated, can be uniquely named by supplying a qualified path from the map down to the required element. The construct *Variable* referenced in the *InclusionOrExclusion* is the qualified path to such an element. Because the `include` and `exclude` constructs are processed in order the resolution of ambiguity can be obtained with just a few `include` and `exclude` clauses.

For example, if `A` is the aggregate item containing items `B`, `C` and `D` and `E`; and if only `A.C` is present in a particular type, then this can be specified in the following manner:

```
exclude A    -- this excludes A and all its children, i.e. B, C, D and E.
include A.C  -- this includes C and its parent A.
```

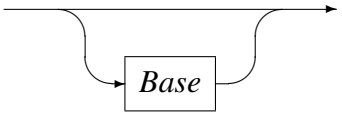
OptCondition



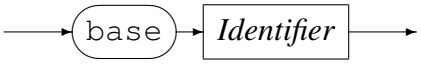
The include and exclude clauses can be conditionally applied. If an include or exclude clause is prefixed with the *Condition* and a particular buffer results in a true evaluation of the Boolean valued *Expression* then the item will be included if the *Condition* prefixes an include clause, and if the *Expression* evaluates to false the item will be excluded. If the *Condition* prefixes an exclude clause, then the opposite behaviour will result.

> **Implementation note:** While syntactically valid, the present implementation does not evaluate the Boolean-valued *Expression* and the item is unconditionally included or excluded depending on the presence of the include or exclude clause.

OptBase



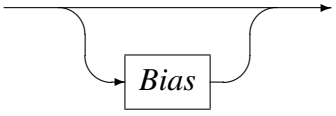
Base



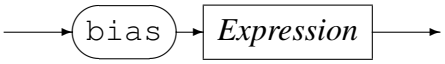
Some of the elements required in the determination of the *Type* of an object may not be present in that object's buffer. For example, if a record had a particular type because it had a sequence number which was not contained within the record itself, then the type *Type* definition can reference an externally named map. The *Base* construct indicates that the corresponding *Map* is external to the object being described. The following example illustrates this:

```
map control\_data base control
```

OptBias



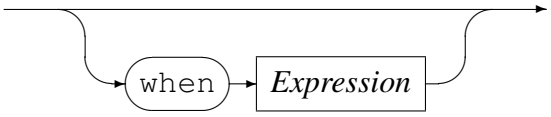
Bias



The base of the offsets in a meta-data map are zero-based and their position in the object being defined might be offset from this by some value which may be a constant or may be a variable offset by some amount known only by considering the current object. This bias value is used to adjust all offsets in the *map* and allows different portions of an object to be mapped by different and independent maps.

```
map test\_map\_a\_trailer include test\_map\_a\_trailer bias 11
```

OptPredicate



If the `when` clause is present in the *OptPredicate* construct then a given object has the corresponding *Type* if the Boolean-valued expression evaluates to true. If the *OptPredicate* is empty then the expression is assumed to evaluate to true. The `objtypes` library evaluates this expression with a given object type when determining the *Type* of an object. The Boolean-valued *Expression* can be over any of the elementary elements of the listed *Maps* whether or not those elements are covered by `include` or `exclude` clauses, in particular, elementary items may come from `based` *Map*s. This is illustrated in the following example:

```
type types\_a
   title "First sample type member --- record type A"
   book controls, book testbook
   map control\_data base control
   map test\_map\_a include test\_map\_a
   map test\_map\_a\_trailer include test\_map\_a\_trailer bias 11
   when (control\_data.direction = 'I')
      and (test\_map\_a.ra\_record\_type\_a = "A");
```

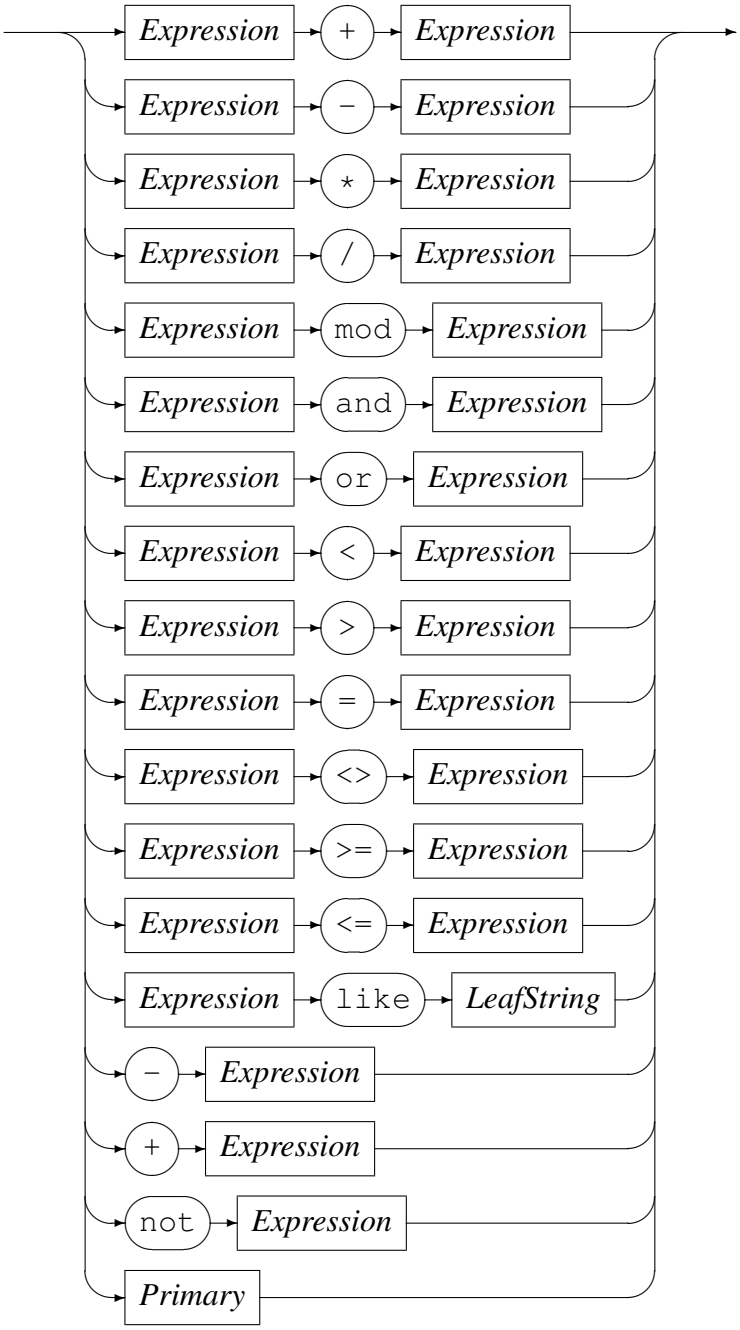
In this example, the book `control` could look something like:

```
01 control\_data.
   03 direction pic x.
```

and the book `testbook` could look something like:

```
01  test-map-a.
    03 ra-record-type-a        pic x value "A".
    03 ra-test-field-1         pic x(4).
       88 ra-test-field-1-cond value "AAAA".
    03 ra-test-field-2         pic x(1).
    03 ra-test-field-3         pic s9(4) comp.
    03 ra-test-field-4         pic s9(4) comp-3.
01  test-map-a-trailer.
    03 ra-test-field-5 occurs 3.
       05 ra-test-subfield-5  pic s9(4) comp occurs 3.
01  test-map-b.
    03 rb-record-type-b        pic x value "B".
    03 rb-test-field-1         pic x(4).
       88 rb-test-field-1-cond value "BBBB".
    03 rb-test-field-2         pic x(1).
    03 rb-test-field-3         pic s9(4) comp.
    03 rb-test-field-4         pic s9(4) comp-3.
01  test-map-b-trailer.
    03 rb-test-field-5 occurs 3.
       05 rb-test-subfield-5  pic s9(4) comp occurs 3.
```

Expression



Wherever an *Expression* is admissible in a *Type* definition it can be an arbitrary *Expression*, provided it is syntactically correct, as described here, and that the result type of the *Expression* has the expected type (Boolean or integer valued). The restrictions on the sub-expressions in these expressions only needs to satisfy the types of sub-expression required by the specified operators.

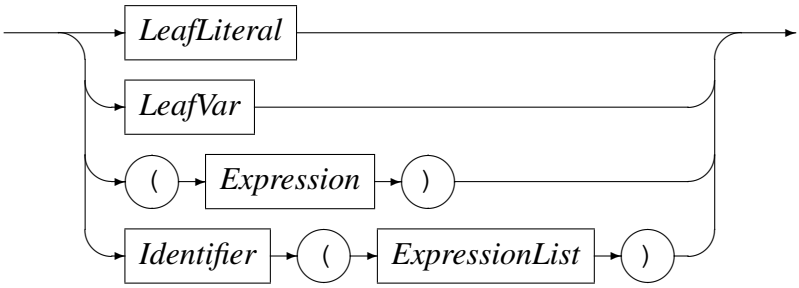
Associates	Operators
left	=, <, >, <>, <=, >=, in
left	+, −, or
left	*, /, div, and, mod
right	not
right	unary +, unary −

Table 1: Operator precedence and associativity

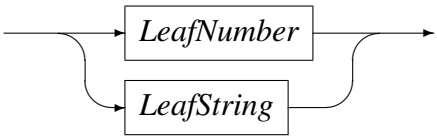
Ambiguity in the evaluation order of operands within an *Expression* is resolved by considering the relative precedence of the operators concerned. The `like` operator is non-associative and the right operand of this operator must be a string literal. This string literal is interpreted as a regular expression. The `like` operator is Boolean valued, evaluating to true if left-hand operand (which is expected to be a string) matches the regular expression; otherwise the `like` operator evaluates to false.

The other operators and their precedence is given in Table 1. The precedence increases going down the table.

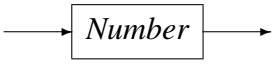
Primary



LeafLiteral

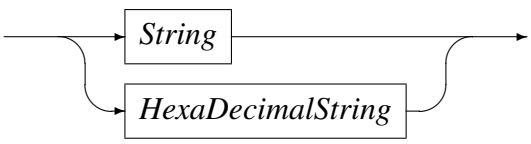


LeafNumber



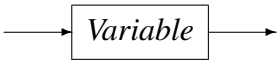
A *Number* has the format described in Section 2.1.1.

LeafString

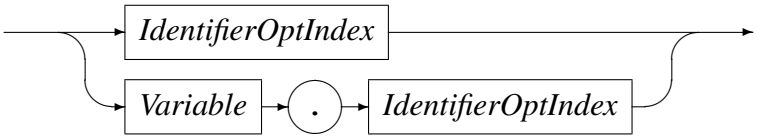


The *String* and *HexaDecimalString* formats are described in Section 2.1.1.

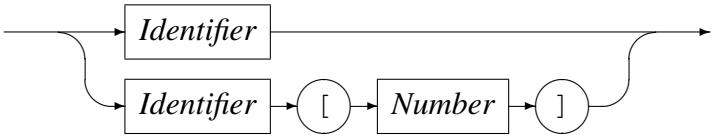
LeafVar



Variable

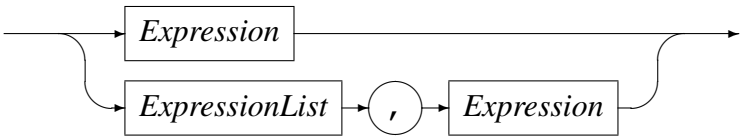


IdentifierOptIndex



A *Variable* is a dot-separated list of *Identifier*s. The list forms a path from a root element *Identifier* (for example a *Map* name) followed by a '.' and an immediate child of that *Identifier*, and so on. Each *Identifier* is optionally followed by a constant index between square brackets, which denotes the specific item amongst an array of multiply occurring items. This is the means by which all elementary items referenced in *Expressions*; and the means by which aggregate and elementary items are referenced in `include` and `exclude` clauses.

ExpressionList



3 Example

This document concludes with a full, but fictitious, example. The example, describes a sequence which is supplied by a sample program. In this example, a named buffer is used to assist the determination of objects of various types as the content of the buffer alone cannot be used to determine the particular object type:

```
-- File: testtype.objtypes
--
-- Test object type specification processed by testprog.c. This collection
-- of types is required by the testprog.c program to test the type
-- collection library objtypes.
--
-- Author: Stephen R. Donaldson.
```

```
--
-- Copyright (c) 2004 Code Magus Limited. All rights reserved.
--
-- $Author: hayward $
-- $Date: 2018/10/19 14:35:58 $
-- $Id: objtpuref.tex,v 1.13 2018/10/19 14:35:58 hayward Exp $
-- $Name:  $
-- $Revision: 1.13 $
-- $State: Exp $
--
-- $Log: objtpuref.tex,v $
-- Revision 1.13  2018/10/19 14:35:58  hayward
-- Correct broken links (take 2)
--
-- Revision 1.12  2018/10/19 14:12:50  release
-- Sort out missing bibliography statement.
--
-- Revision 1.11  2016/05/18 16:28:44  stephen
-- Cleanup example in document
--
-- Revision 1.10  2016/04/02 10:00:37  stephen
-- Add flag value OBJTPFL_NULLS_NA to flags in objtypes_open() and add
-- option nulls_as_na to options in objtypes configuration to set the
-- flag from an objtypes file. Add code to replace null formatted strings
-- with NA when formatting a field.
--
-- Revision 1.9  2011/07/04 09:48:22  hayward
-- Make sure that \cite{} commands are
-- followed by a non breaking space.
--
-- Revision 1.8  2010/01/25 09:59:56  hayward
-- Add rail to Make and fix lines
-- overlapping the right hand margin.
--
-- Revision 1.7  2009/11/30 13:54:53  hayward
-- Change to new title page.
--
-- Revision 1.6  2009/01/16 12:05:25  stephen
-- Add support for environment variable setting from objtypes file.
--
-- Revision 1.5  2005/03/30 21:19:27  stephen
-- Add support for inline constant indexing of variables
--
-- Revision 1.4  2005/03/29 13:32:40  stephen
-- ASCII based machine rebuild
--
-- Revision 1.3  2004/10/31 14:21:20  stephen
-- Rebuild and remove tabs from source
--
-- Revision 1.2  2004/10/17 15:42:10  stephen
-- User reference doc correction
--
```

```
  -- Revision 1.9  2004/06/30 21:37:39  stephen
  -- Allow long identifier names
  --
  -- Revision 1.8  2004/06/17 19:12:26  stephen
  -- Updates from May/June JHB
  --
  -- Revision 1.6  2004/05/19 05:00:52  stephen
  -- Locate leaf node on name and index. Also maintain actual type length
  --
  -- Revision 1.5  2004/05/18 11:18:02  stephen
  -- Windows changes
  --
  -- Revision 1.4  2004/05/10 20:41:38  stephen
  -- Add base buffers to object types library
  --
  -- Revision 1.3  2004/04/19 22:29:30  stephen
  -- New insert to take maps into account for buffer position
  --
  -- Revision 1.2  2004/03/15 18:53:54  stephen
  -- Fix byacc options and testprog must check types returned
  --
  -- Revision 1.1  2004/01/02 22:18:30  stephen
  -- Further development and test files
  --

-- path "%s.cbb";
path "/home/stephen/objtypes/%s.cbb";
-- options endian_little, ascii;

type types_a
   title "First sample type member --- record type A"
   book controls, book testbook
   map control_data base control
   map test_map_a include test_map_a
   map test_map_a_trailer include test_map_a_trailer bias 11
   when (control_data.direction = 'I')
      and (test_map_a.ra_record_type_a = "A");

type types_b
   title "Second sample type member --- record type B"
   book controls, book testbook
   map control_data base control
   map test_map_b include test_map_b
   map test_map_b_trailer include test_map_b_trailer bias 11
   when (control_data.direction = 'I')
      and (test_map_b.rb_record_type_b = "B");

type types_a_record
   title "First sample type member --- record type A"
   book testbook
   map test_map_a include test_map_a
   map test_map_a_trailer include test_map_a_trailer bias 11
```

```
   when  (test_map_a.ra_record_type_a = "A");

type types_b_record
   title "Second sample type member --- record type B"
   book testbook
   map test_map_b include test_map_b
   map test_map_b_trailer include test_map_b_trailer bias 11
   when (test_map_b.rb_record_type_b = "B");
```

References

[1] Code Magus Limited. objtypes: Configuring for Object Recognition, Generation and Manipulation. CML Document CML00018-01, Code Magus Limited, July 2008. PDF.

[2] Code Magus Limited. Symbol Table Loading From Copybooks. CML Document CML00039-01, Code Magus Limited, July 2008. PDF.