MASL Reference Manual

Version 1.4

Synopsis

This document is the definitive language specification and reference for MASL.

Table of Contents

List of Figures

List of Tables

List of Examples

Preface

Model Action Specification Language (MASL) is an implementation independent language for specifying processing within the context of executable Unified Modelling Language (xUML) models and Model Driven Architecture (MDA) [Mellor02][OOA96]. MASL is intended as a replacement for the Action Specification Language (ASL) [ASL96].

The ASL was a direct replacement for Action Data Flow Diagrams (ADFDs) and because of this, was found to be not sufficiently powerful enough to specify the solutions to some problems. This sometimes led to very verbose and unmanageable code, with analysts having to embed traditional languages, such as C++ , within their xUML models.

MASL combines the object oriented features of ASL , together with the more powerful program constructs from other OOD/OOP languages, such as Ada [Ada95] and Java [Gosling00]. In addition, the ASL only specified the dynamic processing of an xUML model. That is, the actions within the states and services. The static description of an xUML model was described using a mixture of diagrams and lists. Using MASL, both the static and dynamic parts of xUML models can be specified.

The development of MASL has been influenced by the revisions and clarification provided in both Project Technology's "The OOA96 Report" and [OOA96] Kennedy Carter's technical note "OOA 97" [OOA97]. Project Technology's papers "Data Types in OOA" [Shlaer97] and their description of a text-based action language named SMALL [Shlaer88] were also of great use. MASL uses the naming of modelling artifacts as described in [Shlaer97] and [Shlaer88] rather than the newer UML naming.

MASL increases the importance of the interfaces between xUML models. Most of these ideas come directly from the Common Object Request Broker Architecture (CORBA) [OMG97] and its Interface Definition Language (IDL).

1. Organization of the Manual

The MASL Reference Manual is organized as described in the following paragraphs:

Chapter 1, *Grammars* describes the grammars and notation used to represent the lexical and syntactic grammars for MASL.

Chapter 2, *Lexical Structure* describes the lexical structure of the language.

Chapter 3, *Types* describes MASL types and variables.

Chapter 4, *Type Conversion* describes the type hierarchy formed by the set of types in a program and hence the implicit type conversions that can occur and the explicit type conversions that can be applied.

Chapter 5, *Domains* describes the structure of a system, which is organized into domains.

Chapter 6, *Objects* describes objects and their attributes, services and lifecycles.

Chapter 7, *Relationships* describes the modelling of relationships between objects in a domain.

Chapter 8, *Lifecycles* describes lifecycles of objects.

Chapter 9, *Functions* describes the functions provided by domains and objects.

Chapter 10, *Services* describes the services provided by domains and objects.

Chapter 11, *Exceptions* describes the MASL exception handling mechanisms.

Chapter 12, *Statements* describes MASL statements.

Chapter 13, *Names* describes the MASL naming system and how to determine what names denote.

Chapter 14, *Expressions* describes MASL expressions.

Appendix A, *Language Defined Characteristics* provides a summary of the language characteristics defined elsewhere in the manual.

Appendix B, *Syntax Summary* presents the complete grammar for MASL.

Bibliography

Books

[Ada83] . American National Standards Institute, 1983.

[Ada95] . International Organization for Standardization, 2001. ©1995. ©2001.

[ASCII91] . International Organization for Standardization, 1991. ©1991. ©2002.

[Barnes98] BARNES, John. 2nd Edition. Addison Wesley Professional, 1998. ISBN ISBN 0-201-34293-6.

[CCS92] . International Organization for Standardization, 1992. ©1992. ©2002.

[EBNF96] . International Organization for Standardization, 1996. ©1996. ©2001.

[Gosling00] GOSLING, James, et al. 2nd Edition. Addison Wesley Professional, 2000. ©1996. ©1997. ©1998. ©1999. ©2000. ISBN ISBN 0-201-31008-2.

[Mellor02] MELLOR, Stephen and BALCER, Marc. Addison Wesley Professional., 2002. ISBN ISBN 0-201-74804-5.

[OMG97] . Revision 2.1. Object Management Group, Inc., 1997.

[Shlaer88] SHLAER, Sally and MELLOR, Stephen. Prentice Hall, Inc., 1988. ISBN ISBN 0-13-629023-X.

[Shlaer92] SHLAER, Sally and MELLOR, Stephen. Prentice Hall, Inc., 1992. ISBN ISBN 0-13-629940-7.

Papers

[ASL96] WILKIE, Ian, et al. Version 2.5. Kennedy Carter, 1996. ©1996.

[OOA97] WILKIE, Ian, CARTER, Colin, and FRANCIS, Paul. Issue 1.0. Kennedy Carter, 1997. ©1997.

[OOA96] SHLAER, Sally and LANG, Neil. Version 1.0. Project Technology, Inc., 1996. ©1996.

[ProjTech97] . Project Technology, Inc., 1997. ©1997.

[Shlaer97] SHLAER, Sally and MELLOR, Stephen. Version 971031. Project Technology, Inc., 1997. ©1997.

1. Grammars

This chapter describes the grammars, used to define the lexical and syntactic structure of a program.

A grammar consists of a number of productions. Each production has an abstract symbol called a non-terminal as its left-hand side and a sequence of one or more non-terminal and terminal symbols as its right-hand side. For each grammar, the terminal symbols are drawn from a specified alphabet.

Starting from a sentence consisting of a single distinguished non-terminal called the goal symbol, a given grammar specifies a language, namely, the infinite set of possible sequences of terminal symbols that can result from repeatedly replacing any non-terminal in the sequence with a right-hand side of a production for which the non-terminal is the left-hand side.

1.1. The Lexical Grammar

The lexical grammar is given in Chapter 2, *Lexical Structure*. This grammar has as its terminal symbols the characters of a character set. It defines a set of productions that describe how sequences of characters are translated into sequences of input elements.

These input elements, with white space and comments discarded, form the terminal symbols for the syntactic grammar and are called tokens. These tokens are the identifiers, reserved words, literals, separators and operators of the language.

1.2. The Syntactic Grammar

The syntactic grammar is given in Chapter 3, *Types* to Chapter 13, *Names*. This grammar has tokens defined by the lexical grammar as its terminal symbols. It defines a set of productions, starting from the goal symbol that describes how sequences of tokens can form syntactically correct programs.

[1] goal = compilation unit ;

Compilations units are described in Section 5.1, “Compilation Units”.

1.3. Grammar Notation

The EBNF notation is used for the grammar [EBNF96]. The EBNF notation provides a standardized, simple and unambiguous means of expressing the grammar of a language.

2. Lexical Structure

This chapter defines the lexical structure of the language.

Programs are written using only character set defined in ISO/IEC 646:1991 [CCS92] and the C0 set of ISO/IEC 6429:1992 [ASCII91] this combined character set is more commonly referenced to as ASCII.

Line terminators are defined to support the different conventions of existing host systems while maintaining consistent line numbers.

The character are reduced to a sequence of input elements, which are white space, comments and tokens. The tokens are the identifiers, reserved words, literals, separators and operators of the syntactic grammar.

2.1. Line Terminators

The definition of lines determines the line numbers produced by a compiler or other component. It also specifies the termination of a single line comment.

[2] line terminator = ? ISO 6429 Line Feed ? | ? ISO 6429 Carriage Return ? | ? ISO 6429 Carriage Return ?, ? ISO 6429 Line Feed ? ;

Lines are terminated by the ASCII characters 'Carriage Return', 'Line Feed' or 'Carriage Return' 'Line Feed'.

2.2. White Space

White space is defined as the ASCII space, horizontal tab and form feed, as well as the line terminators.

[3] white space = ' ? ISO 646 Space character ?' | ? ISO 6429 Horizontal Tabulation ? | ? ISO 6429 Form Feed ? | line terminator ;

2.3. Comments

The language defines only one kind of comment; a single line comment. All the text from the ASCII characters `//` to the end of the line are ignored.

[4] single line comment = '//', {comment character}, line terminator ;

[5] letter = 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j' | 'k' | 'l' | 'm' | 'n' | 'o' | 'p' | 'q' | 'r' | 's' | 't' | 'u' | 'v' | 'w' | 'x' | 'y' | 'z' | 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' | 'J' | 'K' | 'L' | 'M' | 'N' | 'O' | 'P' | 'Q' | 'R' | 'S' | 'T' | 'U' | 'V' | 'W' | 'X' | 'Y' | 'Z' ;

[6] digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' ;

[7] other character = '!' | '"' | '#' | '$' | '%' | '&' | '''' | '(' | ')' | '*' | '+' | ',' | '-' | '.' | '/' | ':' | ';' | '<' | '=' | '>' | '?' | '@' | '[' | '\' | ']' | '^' | '_' | '`' | '{' | '|' | '}' | '~' ;

[8] comment character = letter | digit | other character | white space ;

The lexical grammar implies that comments do not occur within character literals or string literals.

2.4. Identifiers

An identifier is an unlimited-length sequence of letters, digits and underscores, the first of which must be a letter. An identifier cannot have the same spelling as a reserved word.

In addition, an identifier cannot look like a relationship number.

[9] identifier = letter {(letter | digit | underscore)} ;

[5] letter = 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j' | 'k' | 'l' | 'm' | 'n' | 'o' | 'p' | 'q' | 'r' | 's' | 't' | 'u' | 'v' | 'w' | 'x' | 'y' | 'z' | 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' | 'J' | 'K' | 'L' | 'M' | 'N' | 'O' | 'P' | 'Q' | 'R' | 'S' | 'T' | 'U' | 'V' | 'W' | 'X' | 'Y' | 'Z' ;

[6] digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' ;

[10] underscore = '_' ;

The letters include upper-case and lower-case ASCII letters.

The digits include the ASCII digits 0 to 9. Two identifiers are the same only if they are identical, that is, have the same character for each letter or digit. Identifiers are case sensitive, that is, upper-case and lower-case letters are not treated as the same letter.

2.4.1. Relationship Numbers

[11] relationship number = 'R', non zero digit, {digit} ;

[12] non zero digit = digit - '0' ;

2.5. Reserved Words

The following ASCII character sequences are reserved for use as reserved words and cannot be used as identifiers.

[13] reserved words = 'abs' | 'and' | 'any' | 'array' | 'assigner' | 'associate' | 'bag' | 'begin' | 'boolean' | 'byte' | 'Cannot_Happen' | 'case' | 'character' | 'conditionally' | 'console' | 'create' | 'creation' | 'Current_State' | 'declare' | 'deferred' | 'delay' | 'delete' | 'delta' | 'device' | 'digits' | 'disunion' | 'domain' | 'elements' | 'else' | 'elsif' | 'end' | 'enum' | 'event' | 'exception' | 'exit' | 'false' | 'find_all' | 'find_one' | 'find_only' | 'find' | 'for' | 'from' | 'functions' | 'generate' | 'identifier' | 'if' | 'Ignore' | 'in' | 'instance' | 'intersection' | 'is' | 'is_a' | 'link' | 'loop' | 'many' | 'mod' | 'native' | 'Non_Existent' | 'not' | 'not_in' | 'null' | 'numeric' | 'object' | 'of' | 'one' | 'ordered_by' | 'or' | 'others' | 'out' | 'pragma' | 'preferred' | 'private' | 'project' | 'public' | 'raises' | 'raise' | 'range' | 'readonly' | 'referential' | 'relationship' | 'rem' | 'return' | 'reverse_ordered_by' | 'reverse' | 'sequence' | 'service' | 'set' | 'start' | 'state' | 'string' | 'structure' | 'subtype' | 'terminal' | 'then' | 'this' | 'to' | 'transition' | 'true' | 'type' | 'unassociate' | 'unconditionally' | 'union' | 'unique' | 'unlink' | 'using' | 'wcharacter' | 'when' | 'while' | 'with' | 'wstring' | 'xor' ;

Reserved words obey the rules for identifiers, in particular they are case sensitive.

2.6. Literals

A literal represents a value literally, that is, by means of a notation suited to its kind.

[14] literal = numeric literal | character literal | string literal | boolean literal | enumerator literal | null literal | device literal | stream literal ;

2.6.1. Numeric Literals

There are two kinds of numeric literals, decimal literals and based literals.

[15] numeric literal = decimal literal | based literal ;

The type of a numeric literal is numeric.

2.6.1.1. Decimal Literals

A *decimal literal* is a *numeric literal* in the conventional decimal notation (that is, the base is ten).

[16] decimal literal = numeral, ('' | '.', numeral | exponent) ;

[17] numeral = digit, {digit} ;

[18] exponent = ('e'|'E'), [('+'|'-')], numeral ;

A *decimal literal* has the following parts: a whole number part, an optional fractional part and on optional exponent. The fractional part, if present, is indicated by a decimal point (represented by an ASCII period character). The exponent, if present, is indicated by the ASCII letter e or E followed by an optionally signed integer.

An exponent indicates the power of ten by which the value of the *decimal literal* without the exponent is to be multiplied to obtain the value of the *decimal literal* with the exponent.

2.6.1.2. Based Literals

A *based literal* is a *numeric literal* in a form that specifies the base explicitly.

[19] based literal = base, '#', based numeral, ;

[20] base = numeral ;

[21] based numeral = extended digit, {extended digit} ;

[22] extended digit = digit | letter ;

The base (the numeric value of the decimal numeral preceding the #) shall be at least two and at most thirty six. The *extended digits* A through Z represent the digits ten through thirty five, respectively. The value of each extended digit of a *based literal* shall be less then the base.

The extended digits A through Z can be written either in lower-case or upper-case, with the same meaning.

2.6.2. Character Literals

A *character literal* is expressed as a character or an escape sequence, enclosed in single quotes.

[23] character literal = "'", single character, "'" | "'", escape sequence, "'" ;

[24] single character = [(letter | ((other character - "') - '\')] ;

The type of a *character literal* is wcharacter.

2.6.3. String Literals

A *string literal* consists of zero of more characters enclosed in double quotes. Each character may be represented by an escape sequence.

[25] string literal = '"', {string character}, '"' ;

[26] string character = [(letter | escape sequence, ((other character - "") - '\')] ;

The type of a *string literal* is wstring.

2.6.4. Escape Sequences for Character and String Literals

The character and string escape sequences allow for the representation of some non-graphic characters as well as the single quote, double quote and backslash characters in *character literals* and *string literals*.

[27] escape sequence = '\n' | '\r' | '\t' | '\b' | '\f' | '\"' | '\'' | '\\' | octal escape | unicode escape ;

[28] octal escape = '\', octal lead digit, 2 * {octal digit} ;

[29] octal lead digit = '0' | '1' | '2' | '3' ;

[30] octal digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' ;

[31] unicode escape = '\u', extended digit, extended digit, extended digit, extended digit ;

All of the digits in an octal escape must be s digit `0` through `7`. The first digital in an octal escape with three digits must be digit `0` through `3`.

2.6.5. Boolean Literals

The boolean type has two values represented by the reserved words `true` and `false`.

[32] boolean literal = 'false' | 'true' ;

The type of a *boolean literal* is boolean.

2.6.6. Enumerator Literals

Enumerator literals are described in Section 3.7.3, “Enumeration Types”.

2.6.7. The Null Literal

[33] null literal = 'null' ;

The type of *null literal*, represented by the reserved word `null`, is the untyped instance type.

2.6.8. Device Literal

A *device literal* denotes some standard I/O device stream. The only device that is currently supported is `console` which equates to the standard input and output.

[34] device literal = 'console' ;

2.6.9. Stream Literals

The *stream literals* control the output to devices; either user defined devices or the standard console device.

[35] stream literal = 'endl' | 'flush' ;

The `endl` appends a newline to the buffered data stream and then flushes the output to the device. The `flush` flushes the buffered stream to the device.

2.7. Separators

The following tokens are the separators (punctuators).

[36] separator = '"' | '(' | ')' | '|' | ',' | '.' |'..' | ':' | '::' | ':=' | ';' | '=>' | '[' | ']' ;

2.8. Operators

The following tokens are the operators.

[37] operator = '*' | '**' | '+' | '-' | '&' | '->' | '/' | '/=' | '<' | '<=' | '=' | '>' | '>=' | '>>' | '<<' ;

2.9. Pragmas

A pragma is a compiler directive. There are language defined pragmas that give instructions for optimization, listing control etc. Pragmas are only allowed after a semicolon delimiter.

The name of a *pragma* is the identifier following the reserved word `pragma`. The following sequence of identifiers and literals are the pragma values.

[38] pragma list = {pragma, ';'} ;

[39] pragma = 'pragma', pragma name, '(', pragma value list, ')' ;

[40] pragma name = identifier ;

[41] pragma value list = pragma value, {(',', pragma value)} ;

[42] pragma value = identifier | literal ;

3. Types

MASL is a strongly typed language, which means that every variable, attribute, parameter and expression has a type that is known at compile-time. Types limit the values that a variable, attributed and parameter can hold, or that an expression can produce, limiting the operations supported on those values and determine the meaning of the operations.

A type is characterized by a set of values and a set of operations, which implements the fundamental aspects of its semantics. An entity of a given type is a run-time entity that contains a value of the type.

3.1. Type Specifications

The type of an entity is given by a type specification.

A type specification is given in terms of a built-in type, a *collection type*, an instance type or a user *defined type*.

[43] type = built-in type | collection type | instance type | user defined type ;

3.2. Built-in Types

A built-in type is pre-defined by the language and named by its reserved word.

[44] built-in type = 'character' | 'wcharacter' | 'boolean' | 'byte' | 'string' | 'wstring' | 'device' | 'enum' | 'event' | 'instance' ;

Notice that there are no built-in numeric types. Numeric types are discussed in Section 3.7.1, “Numeric Types”.

3.2.1. Character Type

The character type is an 8-bit quantity which encodes a single-byte character from any byte-oriented code set.

The features of the `character` type are the same as the features of the wcharacter type (see Section 3.2.2, “Wide Character Type”).

3.2.2. Wide Character Type

The `wcharacter` type encodes wide characters from any character set.

The features of the `wcharacter` type are given in Table 3.1, “Wide Character Type Features”, where `c` and `c2` represent entities of type on which the feature is applied.

Table 3.1. Wide Character Type Features

Feature	Description
=	Returns `true` if c is equal to c2.
/=	Returns `true` if c is not equal to c2.
<	Returns `true` if c is less than c2.
>	Returns `true` if c is greater than c2.
<=	Returns `true` if c is less than or equal to c2.
>=	Returns `true` if c is greater than or equal to c2.

3.2.3. Boolean Type

The `boolean` type is used to denote a data item that can only take one of the values `true` or `false`.
The features of the `boolean` type are given in Table 3.2, “Boolean Type Features”, where b and b2 represent entities of the type on which the feature is applied.

Table 3.2. Boolean Type Features

Feature	Description
=	Returns `true` if b is equal to b2.
/=	Returns `true` if b is not equal to b2.
and	Returns `true` if both b and b2 are `true`.
xor	Returns `true` if either b or b2 is `true`, but not both.
or	Returns `true` if either b or b2 is `true`.
not	Returns `true` if either b is `false`.

3.2.4. Byte Type

The byte type is an 8-bit quantity that is guaranteed not to undergo any conversion when transmitted by the communication system.
The features of the byte type are the same as the features of any numeric type (see Section 3.7.1, “Numeric Types”).

3.2.5. String Type

The string type is similar to a sequence of character. Strings are singled out as a separate type because they have certain special properties that make them distinct.
The features of the string type are the same as the features of the `wstring` type (see Section 3.2.6, “Wide String Type”).

3.2.6. Wide String Type

The wstring type encodes wide characters from any character set.

The features of the wcharacter type is similar to a sequence of wcharacter. Wide strings are singled out as a separate type because they have certain special properties that make them distinct from a sequence of wcharacter

The features of the wstring are given in Table 3.3, “Wide String Type Features”, where `s` and `s2` represent entities of the type on which the feature is applied.

Table 3.3. Wide String Type Features

Feature	Description
`=`	Returns `true` if `s` is equal to `s2`.
`/=`	Returns `true` if `s` is not equal to `s2`.
`<`	Returns `true` if `s` is less than `s2`.
`>`	Returns `true` if `s` is greater than `s2`.
`<=`	Returns `true` if `s` is less than or equal to `s2`.
`>=`	Returns `true` if `s` is greater than or equal to `s2`.
`&`	Returns the concatenation of `s2` to end of `s`.
`[]`	Returns either the element of `s` at a specific index or the slice of `s` given by a range in indices.

3.2.7. Device Type

The device type denotes a user defined input/output device stream. The features of the type are fully described in Section 12.6, “Output Stream”, Section 12.7, “Input Stream” and Section 12.8, “Input Line Stream”.

3.2.8. Anonymous Enum Type

The enum type is the basis of all enumeration types. Used to pass anonymous enumeration values in the Text_IO domain.

> **Caution**
>
> The anonymous enum type is a deprecated language feature and will be removed in a future version of MASL.

3.2.9. Anonymous Event Type

The anonymous event type is the type of all events. It is used to pass events in the SM_TIMER domain.

Caution

The anonymous event type is a deprecated language feature and will be removed in a future version of MASL.

3.2.10. Anonymous Instance Type

The anonymous instance type is the type of all instance types. Used to pass anonymous instances in the SM_TIMER domain.

Caution

The anonymous instance type is a deprecated language feature and will be removed in a future version of MASL.

3.3. Collection Types

A *collection type* is a type whose values consist of zero or more elements all of which have the same type. Four forms of collection types are provided; sets, bags, sequences and arrays.

[45] collection type = set type | bag type | sequence type | array type ;

A set is a mathematical set. It does not contain duplicate elements. A bag is a like a set, which may contain duplicates. A sequence is like a bag in which elements are ordered. Both bags and sets have no order defined on them. An `array` is like a sequence except that the number of elements in the array is fixed at compile-time.

The features of the each of the *collection type* is given in Table 3.4, “Collection Type Features”.

Table 3.4. Collection Type Features

Collection Type	Maintains Insertion Order	Allows Duplicates	Fixed Length
set	No	No	No
bag	No	Yes	No
sequence	Yes	Yes	No
array	Yes	Yes	Yes

3.3.1. Set Types

A set type is a *collection type* whose values do not contain duplicate elements and which have no order defined on them.

[46] set type = 'set of', type ;

The features of a set type are given in Table 3.5, “Set Type Features”, where s and s2 represent entities of the type on which the feature is applied.

The features ordered_by and reverse_ordered_by are only defined for set types whose element type is an instance type.

The feature -> is only defined for set types whose element type is an instance type.

Table 3.5. Set Type Features

Feature	Description
union	Returns the union of s and s2. The result set contains all the elements in either set.
intersection	Returns the intersection of s and s2. The result set contains all the elements common to both sets.
not_in	Returns the set containing all the elements in s that are not in s2.
disunion	Returns the set containing all the elements in either s or s2, but not in both.
&	Returns the set containing all the elements in s and s2.
ordered_by	Returns a sequence that contains all the elements of s in specified ascending order.
reverse_ordered_by	Returns a sequence that contains all the elements of s in specified descending order.
find	Returns a set containing all the elements of s that satisfies the specified condition.
find_all	Returns a set containing all the elements of s.
find_one	Returns an arbitrary element of s that satisfies the specified condition.
find_only	Returns the only element of s that satisfies the specified condition.
->	Returns the bag of elements that are related to the elements of s using the specified relationship.

3.3.2. Bag Types

A bag type is a *collection type* whose values can contain duplicate elements which have no order defined on them.

[47] bag type = 'bag of', type ;

The features of a bag type are give in Table 3.6, “Bag Type Features”, where b and b2 represent entities of the type on which the feature is applied.

The features `ordered_by` and `reverse_ordered_by` are only defined for bag types whose element type is either a structure or an instance type.

Table 3.6. Bag Type Features

Feature	Description
`union`	Returns the union of b and b2. The result set contains all the elements in either bag.
`intersection`	Returns the intersection of b and b2. The result set contains all the elements that are in both bags.
`not_in`	Returns the bag containing all the elements in b not in b2.
`disunion`	Returns the bag containing all the elements in either b or b2, but not in both.
`&`	Returns the bag containing all the elements in b and b2.
`ordered_by`	Returns a sequence that contains all the elements of b in specified ascending order.
`reverse_ordered_by`	Returns a sequence that contains all the elements of b in specified descending order.
`find`	Returns a bag that contains all the elements of b that satisfy the specified condition.
`find_all`	Returns a bag that contains all the elements of b.
`find_one`	Returns an arbitrary element of b that satisfies the specified condition.
`find_only`	Returns the only element of b that satisfies the specified condition.
`->`	Returns the bag of elements that are related to the elements of b using the specified relationship.
`get_unique`	Returns a set containing all the elements of b.

The features `find`, `find_all`, `find_one` and `find_only` are only defined for bag types whose element type is an instance type.

The feature -> is only defined for bag types whose element type is an instance type.

3.3.3. Sequence Types

A sequence type is a *collection type* whose value can contain duplicate elements which have order.

[48] sequence type = 'sequence', ['(', expression, ')'], 'of', type ;

Sequences are indexed by values of the pre-defined type positive which is declared in the standard domain.

The features of a sequence type are given in Table 3.7, “Sequence Type Features”, where s and s2 represent entities of the type on which the feature is applied.

The features ordered_by and reserve_ordered_by are only defined for sequence types whose element type is either a structure or an instance type.

The features find, find_all, find_one and find_only are only defined for sequence types whose element is an instance type.

The feature -> is only defined for sequence types whose element is an instance type.

The optional expression defines the upper bound of the sequence and must equate to a positive qualified type.

Table 3.7. Sequence Type Features

Feature	Description
&	Returns the set containing all the elements in q and q2.
[]	Returns the either the element of q at a specific index or the slice of q given by a range of indices.
ordered_by	Returns a sequence that contains all the elements of s in specified ascending order.
reverse_ordered_by	Returns a sequence that contains all the elements of s in specified descending order.
find	Returns a bag that contains all the elements of q that satisfy the specified condition.
find_all	Returns a bag that contains all the elements of q.
find_one	Returns an arbitrary element of q that satisfies the specified condition.

Feature	Description
`find_only`	Returns the only element of q that satisfies the specified condition.
`->`	Returns the bag of elements that are related to the elements of q using the specified relationship.

3.3.4. Array Types

An array type is a *collection type* whose values can contain duplicate elements which have no order and has a fixed number of elements.

[49] array type = 'array', '(', range, ')', 'of', type ;

Arrays are indexed by values given by the type of its range. This range must resolve to some specific integer or enumeration type. If the type of the range resolves to numeric type, then the range is defined to be of the pre-defined type integer with bounds given by a conversion to integer of the bounds of the range.

The features of an array type are given in Table 3.8, “Array Type Features”, where a represents the instance on which the feature type is applied.

The features `ordered_by` and `reserve_ordered_by` are only defined for array types whose element type is either a structure or an instance type.

The features `find`, `find_all`, `find_one` and `find_only` are only defined for array types whose element is an instance type.

The feature `->` is only defined for array types whose element is an instance type.

Table 3.8. Array Type Features

Feature	Description
`[]`	Returns the either the element of a at a specific index or the slice of a given by a range of indices.
`ordered_by`	Returns a sequence that contains all the elements of a in specified ascending order.
`reverse_ordered_by`	Returns a sequence that contains all the elements of a in specified descending order.
`find`	Returns a set that contains all the elements of a that satisfy the specified condition.
`find_all`	Returns a set that contains all the elements of a.
`find_one`	Returns an arbitrary element of a that satisfies the specified condition.

Feature	Description
`find_only`	Returns the only element of a that satisfies the specified condition.
`->`	Returns the sequence of elements that are related to the elements of a using the specified relationship.
`get_unique`	Returns a set containing all the elements of the array.

3.4. Instance Types

An instance type is a type that refers to an instance of a given object.

[50] instance type = 'instance', 'of', scoped name ;

A compile-time error occurs if the scoped name does not name an object that is visible to the enclosing domain.

The features of an instance type are given in Table 3.9, “Instance Type Features”, where `i` and `i2` represent instances on which the feature is applied and `0` represents the object.

Table 3.9. Instance Type Features

Feature	Description
`=`	Returns `true` if `i` is equal to `i2`.
`/=`	Returns `true` if `i` is not equal to `i2`.
`->`	Returns either the instance or bag of instances that are related to `i` using the specified relationship.
`with_->`	Returns either the associative instance that models a specific relationship between `i` and `i2`.
`find`	Returns the set that contains all the instances of `0` that satisfy a condition.
`find_all`	Returns a set containing all the instances of `0`.
`find_one`	Returns an arbitrary instance of `0` that satisfies the specified condition.
`find_only`	Returns the only instance of `0` that satisfies the specified condition.

3.5. User Defined Types

A *user defined type* is a reference to a type that has already been declared. Types defined in the current domain can be named using just their type name. Types from

other domains can be accessed by using a scoped name. Only public types from other domains are accessible. Use of this qualified form, where the type is in the enclosing domain, is allowed.

[51] user defined type = scoped name ;

A compile-time error occurs if the scoped name does not name a type that is visible to the enclosing domain.

3.6. Type Declarations

A *type declaration* provides the analyst with a way to construct new user defined types. There are two forms of *type declaration*, full type declarations and subtype declarations.

[52] type declaration = type modifier, 'type', type name, 'is' type definition, ';' | type modifier, 'subtype', type name, 'is' subtype definition ;

[53] type name = identifier ;

A compile-time error occurs if the type name naming a type is already declared as a type in the enclosing domain.

3.6.1. Type Modifiers

[54] type modifier = [('public' | 'private')] ;

3.6.1.1. Public Types

If a type is declared public, then it may be accessed by any code that can access the domain in which it is declared.

A compile-time error occurs if a public type or subtype is declared in terms of some private type.

3.6.1.2. Private Types

If a type is declared private, then access is permitted only when it occurs from within the domain in which it is declared.

3.7. Full Type Declarations

A *full type* declaration introduces a new type quite distinct from any other type. The set of values belonging to two distinct types are themselves distinct, although in some cases the actual lexical form of the values may be identical; which one

is meant at any point is determined by the context. Values of one type cannot be assigned to a variable of another type.

A type definition gives the definition of a new type.

[55] type definition = type, [constraint] | numeric type definition | structure type definition | enumeration type definition | unconstrained array type definition ;

A type definition can be given by either an existing type, a numeric type definition, a structure type definition, enumeration type definition or an unconstrained array type definition. A compile-time error occurs if a type is declared in terms of some instance types.

A compile-time error also occurs if the optional constraint defines a range of values that is not a subset of the range of values of the original type. Specifically:

- The range of values given by a range constraint must all be values of the original type.
- A digits constraint can only be given if the original type has a digits constraint. In addition the value of the digits constraint must be less than or equal to the original digits constraint.
- A delta constraint can only be given if the original type has a delta constraint.
- The value of the delta constraint must be less than or equal to the original delta constraint.

In practice, the optional constraint can only be given for types of numeric type, enumeration and unconstrained array types.

Note that entities of the new type and the existing type cannot be mixed. Hence if one variable is declared as having its type *this_new_type*, where *this_new_type* is derived from character and another is declared as having the type character, the value of the first cannot be assigned to the value of the second and vice-versa. The features of the new type are that of the existing type.

For unconstrained array type definitions, the bounds of the constraint must satisfy the bounds of the unconstrained type.

3.7.1. Numeric Types

The numeric type definition defines a numeric type that supports data elements that can be used in arithmetic computation. A numeric type can be used to specify either integer or floating-point values.

[56]	numeric type definition	=	'numeric', constraint ;
[57]	constraint	=	range constraint | digits constraint | delta constraint ;
[58]	range constraint	=	'range', range ;
[59]	range	=	expression, '..', expression | name, '', 'range', [('(', ')')] | name, '', 'elements', [('(', ')')] | type, '', 'elements', [('(', ')')] | ;
[60]	digits constraint	=	'digits', constant expression, range constraint ;
[61]	delta constraint	=	'delta', constant expression, range constraint ;

The range of value that numeric type can take is specified with a range constraint. A compile-time error occurs if:

- For the definition of integer types, the range does not specify a range of integer values.
- For the definition of floating-point types, the range does not specify a range of floating-point values.
- The type of the constant expression in a digits constraint is not an integer type. In addition its value must be greater than 0.
- The type of the constant expression in a delta constraint is not a floating-point type. In addition its value must be greater then 0.

The features of a numeric type are given in Table 3.10, “Numeric Type Features”, where n and n2 represent entities of the type on which the feature is applied.

Table 3.10. Numeric Type Features

Feature	Description
=	Returns true if n is equal to n2.
/=	Returns true if n is not equal to n2.
<	Returns true if n is less than n2.
>	Returns true if n is greater than n2.
<=	Returns true if n is less than or equal to n2.
>=	Returns true if n is greater than or equal to n2.
+	Returns the sum of n and n2.

Feature	Description
-	Returns the value of subtracting n2 from n.
*	Returns the value of multiplying n and n2.
/	Returns the value of dividing n from n2.
mod	Returns the modulus n and n2.
rem	Returns the remainder n and n2.
**	Returns the value of raising n to the power of n2.
..	Returns the sequence containing all the values starting with n up to n2, inclusive, in specified ascending order.
+	Returns the positive value of n.
-	Returns the arithmetic negation of n.
abs	Returns the absolute value of n.

The features mod, rem and .. are only defined for numeric types that are integer types.

3.7.2. Structure Types

A structure is a composite entity consisting of named components which may be of different types.

[62] structure type definition = 'structure', component, {component} ;

[63] component = component name, ':' type, [(':=', constant expression], ';' ;

[64] component name = identifier ;

The *component name* in a component declaration may be used to refer to the component.

A compile-time error occurs if:

- A structure declaration contains two components with the same name.
- A public structure type has a component whose type is private.
- The type of any optional constant expression in the declaration of a component is not assignable to the type of the component.

If a component contains a constant expression, then it has the semantics of an assignment to the declared component and the expression is evaluated and the assignment performed each time an instance of the structure type is created.

3.7.3. Enumeration Types

An enumeration type definition defines a new distinct type together with the values of the type.

[65]	enumeration type definition	=	'enum', '(', enumerator, [(',', enumerator)], ')' ;
[66]	enumerator	=	enumerator name, ['=', constant expression] ;
[67]	enumerator name	=	identifier ;
[68]	enumerator literal	=	enumerator name | type name, '.', enumerator name | domain name, '::', enumerator name | domain name, '::', type name, '.', enumerator name ;

A compile-time error occurs if:

- The type of the constraint expression does not specify an integer value.
- An enumeration type declaration contains two enumerators with the same name.
- The optional numeric values given to enumerators in an enumeration type declaration are not in specified ascending order. Note this also means that two enumerators cannot have the same numeric value.

If the same enumerator name is specified for more than one enumeration type definition in the same domain, the corresponding enumeration literals are said to be over-loaded. At any place where an over-loaded enumeration literal occurs, the type of the enumeration literal has to be determined from the context (see Section 14.4.1, “Literals”).

The features of an enumeration type are given in Table 3.11, “Enumeration Type Features”, where e and e2 represent entities of the type on which the feature is applied.

Table 3.11. Enumeration Type Features

Feature	Description
=	Returns true if e is equal to e2.
=	Returns true if e is not equal to e2.

Feature	Description
<	Returns true if e is less than e2.
>	Returns true if e is greater than e2.
<=	Returns true if e is less than or equal to e2.
>=	Returns true if e is greater than or equal to e2.
`..`	Returns the sequence containing all the values starting with e up to e2, inclusive, in specified ascending order.

3.7.4. Unconstrained Array Types

An *unconstrained array type* declaration follows the same rules as defined in Section 3.3.4, “Array Types”, except that the bounds of the range are unconstrained.

[69] unconstrained array type definition = 'array', '(', type, 'range', '<>', ')', 'of', type, ';' ;

The bounds of the array object are determined by:

- A type or subtype declaration which constrain the bounds at the point of declaration.
- A formal parameter, whereby the constraints of the array are obtained from the corresponding actual parameter.

A compile-time error will be raised when:

- An attempt is made to declare a variable of an *unconstrained array type*.
- An attempt is made to declare a parameter of an *unconstrained array type* and who's mode is `out`.

3.8. Subtype Declarations

A `subtype` declaration introduces a type that is characterized by a set of values, which are a subset of the values of another type.

A `subtype` declaration does not introduce a new distinct type. Hence, assigning the value of a variable declared to have this subtype to a variable having the original type is valid. However, the reverse can be invalid, causing a run-time exception to be raised, since the value of the original type may be outside the range of the subtype.

A subtype definition gives the definition of a new subtype.

[70] subtype definition = user defined type, '(', range, ')', ';' | type, [constraint], ';' ;

A compile-time error occurs if:

- A subtype is declared in terms of some instance type.
- The optional constraint defines a range of values that is not a subset of the range of values of the original type.
- The range is not defined for a user defined type which is an *unconstrained array type* type.

Specifically,

- The range of values given by a range constraint must all be values of the original type.
- A digit constraint can only be given if the original type has a digits constraint. In addition the value of the digits constraint must be less than or equal to the original digits constraint.
- A delta constraint can only be given if the original type has a delta constraint. In addition the value of the delta constraint must be less than or equal to the original delta constraint.

  In practice the optional constraint can only be given for subtypes of numeric type and enumeration types.

3.9. Variable Declarations

A *variable declaration* declares a local variable.

[71] variable declaration = variable name, ':', [modifier], type, [('(', range, ')')], [(':=', [extended expression], ';' ;

[72] variable name = identifier ;

Each *variable declaration* declares one local variable, whose name is given by the *variable name*.

The type of the variable is denoted by the type. There are no restrictions on the type of a variable.

A compile-time error occurs if:

- The type of the optional extended expression is not assignable to the type of the variable.
- A `readonly` variable does not have an initial value given by the optional extended expression.
- The type of the variable is an unconstrained array type which does not impose the optional range constraint.
- The optional range constraint is imposed on a type which is not an unconstrained array type.

3.9.1. Modifiers

[73] modifier = 'readonly' ;

Any attempt to assign to a `readonly` variable results in a compile-time error. Therefore, once a `readonly` variable has been initialized, it always contains the same value.

3.9.2. Scope of Variable Declarations

The scope of a variable declaration is the rest of the block in which the declaration appears.

3.9.3. Hiding of Names by Local Variables

If a name declared as a local variable is already declared as another variable in an outer scope, then that outer declaration is hidden throughout the scope of the local variable. The outer variable can almost always still be accessed using an appropriately qualified name.

3.9.4. Execution of Local Variable Declarations

A local variable declaration is, in some aspects, an executable statement. If a variable declaration has an initialization expression, the expression is evaluated and its value is assigned to the variable.

4. Type Conversion

4.1. Types and their Subtypes

The set of types in a program form a type hierarchy. This type hierarchy defines:

- The implicit type conversions that can occur.
- The explicit type conversions that can be applied.

When an expression of one type is used where an expression of another type is expected.

The first section of this chapter describes the general rules for implicit and explicit type conversions between types in the type hierarchy. The rest of the chapter describes the pre-defined type hierarchy for the built-in types, the collection types and the special case of numeric types.

4.1.1. Types and their Subtypes

If a type is a `subtype` of another `type`, then there is an implicit conversion that occurs when an expression of the `subtype` is used where an expression of the original `type` is expected. In addition, there is an implicit conversion that occurs when an expression of the original `type` is used where an expression of the `subtype` is expected. When this `type` conversion is evaluated, the value of the expression is converted to the corresponding value of the target `type` if any. If there is no value of the target `type` that corresponds to the value, an exception is raised.

A subtype of a `type` that is itself a `subtype` of another, is also a `subtype` of this `type`.

If a `type` is defined in terms of another `type` then an explicit type conversion can be used to convert a value of one `type` into a value of the other. If there is no value of the target `type` that corresponds to the value, an exception is raised.

Example 4.1. Type and Subtype Declarations

```
type A is enum (REDM BLUE, GREEN);
type B is enum (RED, YELLOW, BLUE);
subtype C is A;
subtype D is A;
```

```
type E is A;
subtype F is C;
```

If we declare variables a to f of types A to F respectively, then Example 4.2, “Simple type hierarchy conversions” shows, for each possible assignment between the variables, which conversion is valid.

Note that even though type A and B have literals of the same lexical form, there are no valid type conversions between values of one type to another.

Example 4.2. Simple type hierarchy conversions

```
a := a;     // no conversion required
a := b;     // invalid
a := c;     // implicit conversion
a := d;     // implicit conversion
a := e;     // invalid, an explicit conversion of the form
            // a := A(e) is required
a := f;     // implicit conversion

b := a;     // invalid
b := b;     // no conversion required
b := c;     // invalid
b := d;     // invalid
b := e;     // invalid
b := f;     // invalid

c := a;     // implicit conversion that may raise an exception
c := b;     // invalid
c := c;     // no conversion required
c := d;     // implicit conversion that may raise an exception
c := e;     // invalid, an explicit conversion of the form
            // c := C(e) or c := (A)e is required that may
            // raise an exception
c := f;     // implicit conversion

d := a;     // implicit conversion that may raise an exception
d := b;     // invalid
d := c;     // implicit conversion that may raise an exception
```

```
d := d;     // no conversion required
d := e;     // invalid, an explicit conversion of the form
            // d := D(e) is required that may raise an
            // exception
d := f;     // implicit conversion that may raise an exception

e := a;     // invalid, an explicit conversion of the form
            // e := E(a) is required that may raise an
            // exception
e := b;     // invalid
e := c;     // invalid, an explicit conversion of the form
            // e := E(c) is required that may raise an
            // exception
e := d;     // invalid, an explicit conversion of the form
            // e := E(d) is required that may raise an
            // exception
e := e;     // no conversion required
e := f;     // invalid, an explicit conversion of the form
            // e := E(f) is required that may raise an
            // exception

f := a;     // implicit conversion that may raise an exception
f := b;     // invalid
f := c;     // implicit conversion that may raise an exception
f := d;     // implicit conversion that may raise an exception
f := e;     // invalid, an explicit conversion of the form
            // f := F(e) or f := A(e) is required that may
            // raise an exception
f := f;     // no conversion required
```

4.1.2. Properties of Types

The properties that are defined by a type hierarchy also define the implicit conversion that can occur. The explicit type conversions that can be applied are known as assignable and convertible.
A value of type T is convertible to a type S if T and S share a common type. Converting a value of type T to an entity of type S will not raise an exception if navigating the type hierarchy from type T to this common type passes over the type S. In all other case the conversion may raise an exception.
A value of type T is assignable to a type S if T and S share a common subtype. Assigning a value of type T to an entity of type S will not raise an exception if

navigating the type hierarchy from type T to this common subtype passes over the type S. In all other case the conversion may raise an exception.

4.1.3. Base and Basis Types

The *base type* of a type T is the type obtained by navigating up the type hierarchy as far as possible using only subtype relationships.

The *basis type* of a type T is the type obtained by navigating up the type hierarchy as far as possible using both subtype and type relationships.

Both *base types* and *basis types* are used to describe the types of the values returned by expressions.

4.2. Built-in Types

The *built-in types* have the following type hierarchy:

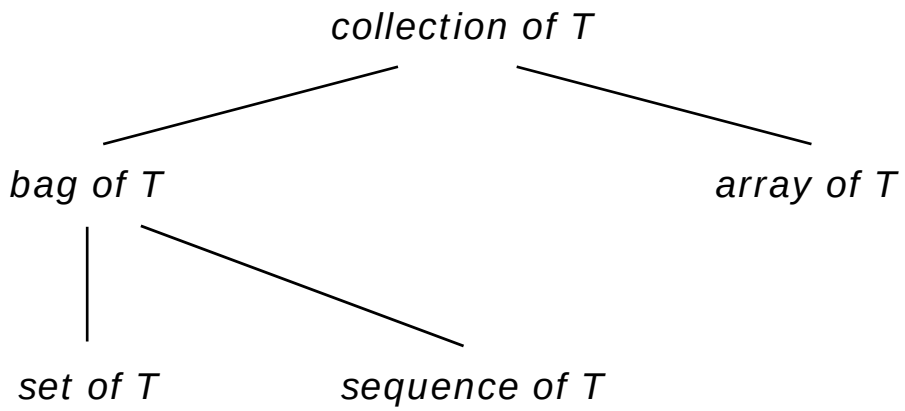
- character is a subtype of wcharacter.
- boolean is not a subtype of another type.
- byte is a type of numeric.
- string is a subtype of wstring

Hence, the value of type character can be used where a value of type wcharacter is expected. In addition, a value of type wcharacter can be used where a value of type character is expected. This second conversion may cause an exception to be raised. Similarly for values of type string and wstring. The numeric type cannot be used a type, but enables type conversions between two numeric types, where one is not a subtype or type of the other. Numeric types are discussed in Section 4.5, “Numeric Types”.

4.3. Collection Types

Collection types of a specific type (T) have the type hierarchy shown in Figure 4.1, “Collection Type Hierarchy”. This means that there are implicit conversions between values of set, bag and sequence types.

Figure 4.1. Collection Type Hierarchy



The collection type cannot be used as a type, but enables type conversions between values of two collection types of the same type.

Values of a collection type can be converted to a different collection type of the same element type by using explicit conversion. This type conversion may raise an exception if the original value cannot be converted to a value of the new type.

Given a is of type array, b is of type bag, s is of type set, q is of type sequence, and *x* and *y* are integer literals (where *x* < *y*). Then Table 4.1, “Conversions for a Collection Type Hierarchy” describes for each possible assignment between variables when an exception will be raised.

Table 4.1. Conversions for a Collection Type Hierarchy

Assignment	**Description**
`s := set of T(b)`	Raises an exception if there are duplicate elements in the bag.

Assignment	Description
`s := set of T(q)`	Raises an exception if there are duplicate elements in the sequence.
`s := set of T(a)`	Raises an exception if there are duplicate elements in the array.
`b := bag of T(s)`	Never raises an exception.
`b := bag of T(q)`	Never raises an exception.
`b := bag of T(a)`	Never raises an exception.
`q := sequence of T(s)`	Never raises an exception. The order of the elements in the sequence will be arbitrary.
`q := sequence of T(b)`	Never raises an exception. The order of the elements in the sequence will be arbitrary.
`q := sequence of T(a)`	Never raises an exception. The order of the elements in the sequence will be the same as the array.
`q := array(`*x*`..`*y*`) of T(s)`	Raises an exception if the number of elements in the set is not the same as the length of the array. The order of the elements in the array will be arbitrary.
`q := array(`*x*`..`*y*`) of T(b)`	Raises an exception if the number of elements in the bag is not the same as the length of the array. The order of the elements in the array will be arbitrary.
`q := array(`*x*`..`*y*`) of T(q)`	Raises an exception if the number of elements in the sequence is not the same as the length of the array. The order of the elements in the array will be the same as the sequence.

To convert a bag, sequence or set without raising an exception, the operation `get_unique` can be used (seeAppendix A, *Language Defined Characteristics*).

4.3.1. Collection Types of Subtypes

A collection type of a subtype of another type is a subtype of the collection type of the original type.

For example, if we declare:

Example 4.3. Collection Type of a Subtype

```
type T is enum (REDM BLUE, GREEN);
subtype S is T;
st:set of T;
ss:set of S;
```

Table 4.2, “Type conversions for Collection Type Hierarchy” describes, for each possible assignment between variables, which conversions are valid.

Table 4.2. Type conversions for Collection Type Hierarchy

Assignment	Description
`st := st`	No conversion required.
`st := ss`	Implicit conversion required.
`ss := st`	Implicit conversion that may raise an exception if any of the element values cannot be converted into a value of type T.
`ss := ss`	No conversion required.

Similarly for `bag`, `sequence` and `array` types.

4.3.2. Collection Element Types

There is a special implicit conversion that can occur between a collection type and its element type.

For example, if we declare:

Example 4.4. Collection Element Type Conversion

```
type T is enum (REDM BLUE, GREEN);
t:T;
s:set of T;
```

```

```

then the following is valid:

```
s := t;
```

Similarly, for bag, sequence types and array types whose length is equal to 1.

4.4. Structure Types

There is a special type conversion that can occur between a type and a structure type that has only one component of that type. For example, if we declare:

Example 4.5. Structure Type, Type Conversion

```
type T is enum (REDM, BLUE,GREEN);

type Structure_type is structure
    c : T;
end structure;
```

then the following is valid:

```
t : T := REDM;
s : Structure_type;
begin
    s := t;
end;
```

4.5. Numeric Types

All *numeric types* are implicit types of the numeric type. The *numeric type* cannot be used in a type declaration, but enables type conversions between values of two *numeric types*, where one is not a subtype or type of the other.

Example 4.6. Numeric Type Declaration

```
type N is numeric range 0 .. 100;
type M is numeric digits 7 range 0 .. 50;
```

Then N is a type of numeric and M is a type of numeric.

If we declare variables n and `m` of types N and M respectively, Table 4.3, “Conversions for a Numeric Hierarchy” describes, for each possible assignment between variables, which conversions are valid.

Converting a value of one numeric type to another only raises an exception if the value is outside the range of the new numeric type. For conversions involving real numeric types, the result is within the accuracy of the specified target type. The conversion of a real value to a integer type rounds to the nearest integer; if the value is halfway between two integers rounding may be either up or down.

Table 4.3. Conversions for a Numeric Hierarchy

Assignment	Description
`n := n;`	No conversion required.
`n := m;`	Invalid, an explicit conversion of the form `n := N(m)` is required, that may raised an exception.
`m := a;`	Invalid, en explicit conversion of the form `m := M(n)` is required, that may raised an exception.
`m := m;`	No conversion required.

5. Domains

In building large software systems, the analyst has to deal with a number of distinctly different subject matters or domains. A domain is a separate world inhabited by a distinct set of objects that behave according to the rules and policies characteristic of the domain.

Domains allow the specification of groups of logically related entities. Typically, a domain contains the declaration of a number of services, which can be called from outside the domain, whilst the inner workings of the domain remain hidden from outside users.

5.1. Compilation Units

A *compilation unit* is the goal symbol for syntactic grammar.

[74] compilation unit = project declaration | domain declaration | domain definition ;

A *compilation unit* is either the declaration of a domain or the definition of a state, domain service or object service.

5.2. Project Declaration

Projects are used to organize the development of applications. A *project declaration* declares a new project. The declaration specifies the name of the project together with declarations of its members, that is, the domains that make up a project.

[75] project declaration = 'project', project name, 'is', {project declarative item}, 'end', ['project'], ';' ;

[76] project name = identifier ;

[77] project declaration item = 'domain', domain name, ';' ;

A compile-time error occurs if the domain named by the *domain name* in a project declaration does not exist.

5.3. Domain Declaration

A *domain declaration* declares a new domain. The declaration specifies the name of the domain together with declarations of its members, that is, types, exceptions, objects, domain services and relationships.

[78] domain declaration = 'domain', domain name, 'is', {domain declarative item}, 'end', ['domain'], ';' ;

[79] domain name = identifier, ;

5.4. Domain Members

The members of a domain are introduced by *domain declarative item*s.

[80] domain declarative item = type declaration | exception declaration | object declaration | domain service declaration | domain function declaration | object predeclaration | relationship declaration ;

Type declarations are described in Section 3.6, “Type Declarations”, exception declarations in Section 11.1, “Exception Declarations”, object declarations in Section 6.1, “Object Declaration”, domain service declarations in Section 10.1, “Domain Services”, domain function declarations in Section 9.1, “Domain Functions”, object pre-declarations in Section 6.2, “Object Pre-declaration” and relationship declarations in Section 7.1, “Relationship Declarations”.

5.5. Domain Definitions

A *domain definition* is either the definition of the block of code that implements a state, an object service or a domain service.

[81] domain definition = state definition | object service definition | domain service definition | domain function definition | object function definition ;

*Domain definition*s are described in Section 8.1.4, “State Definition”,in Section 9.2.6, “Object Function Definition”, in Section 10.2.6, “Object Service Definition”, in Section 9.1.6, “Domain Function Definition” and Section 10.1.6, “Domain Service Definition”.

5.6. Names and Scoping

Identifiers for the following kinds of definition are scoped:

- types
- domain services
- objects
- exceptions
- enumeration values

A name can be used in an unqualified form within a particular scope; it will be resolved by successively searching farther out in enclosing scopes. A qualified name (one of the form `scoped_name::`*identifier*) is resolved by first resolving the qualifier scoped name to a scope S and then locating the definition of *identifier* with S. The identifier must be directly defined in S. The identifier is not searched for in enclosing scopes.

When a qualified name begins with `::`, the resolution starts with current scope and locates subsequent identifiers in the qualified name by the rule described in the previous paragraph. A scoped name is used to refer to entities defined outside of the current scope.

[82] scoped name = identifier | ('::', identifier) | (scoped name, '::', identifier) ;

6. Objects

Objects are entities such that all the instances of the entities have the same characteristics and are subject to and conform to the same set of rules and policies.

Typically, an object contains declarations of the attributes that instances of the object contain, the declarations of services provided by the object and a description of the lifecycle of an instance of the object, if any.

6.1. Object Declaration

An *object declaration* declares a new object. The declaration specifies the name of the object together with the declarations of its members, that is, its attributes, services, events, states, and lifecycles.

[83] object declaration = 'object', object name, 'is', {object declarative item}, 'end', ['object'], ';' ;

[84] object name = identifier ;

Object declarative items are described in Section 6.3, “Object Members”.

A compile-time error occurs if the the object name naming the object has already declared an object in the enclosing domain.

6.2. Object Pre-declaration

An *object pre-declaration* pre-declares a new object. The pre-declaration specifies the name of the object only.

[85] object predeclaration = 'object', object name, ';' ;

An *object pre-declaration* is used to pre-declare an object that has a relationship with other objects in the domain, without declaring its members.

A compile-time errors occurs if the object name naming the object is already pre-declared as an object within the enclosing scope.

6.3. Object Members

The members of an object are introduced by object declarative items.

[86] object declarative item = attribute declaration | object service declaration | object function declaration | event declaration | identifier declaration | state declaration | transition table declaration ;

Attribute declarations are described in Section 6.4, “Attributes”.

6.4. Attributes

The attributes of an object are introduced by *attribute declarations*.

[87] attribute declaration = attribute name, ':', attribute modifier, type, [('(', range , ')'], [(':=', constant expression)], ';' ;

[88] attribute name = identifier ;

The attribute name in an *attribute declaration* may be used to refer to the attribute. A compile-time error occurs if:

- The type of an attribute is in terms of some instance type.
- An object declaration contains two attributes with the same name.
- The type of the optional constant expression in the declaration of an attribute is not assignable to the type of the attribute.
- The optional range constraint is imposed on a type which is not a constrained array type.

The individual attributes of a variable with an instance type as its type can be denoted by following the variable name with a dot and the attribute name.

6.4.1. Initialization of Attributes

It is possible to give defaults for some or all of the attributes.

If an *attribute declaration* contains a constant expression, then it has the semantics of an assignment to the declared attribute and the expression is evaluated and the assignment performed each time an instance of the object is created.

6.4.2. Attribute Modifiers

[89] attribute modifier = ['preferred'], [('referential', relationship formalization)] ;

6.4.2.1. Preferred Attributes

Preferred attributes are part of an objects preferred identifier. The preferred identifier of an object, is the set of one or more of its attributes whose values uniquely distinguish each instance of the object.

[90] identifier declaration = identifier, 'is', '(', attribute name, {(',', attribute name)}, ')', ';' ;

6.4.2.2. Referential Attributes

A *referential attribute* is the formalization of a relationship.

[91] relationship formalization = '(', relationship specification, '.', attribute name, {(',', relationship specification, '.', attribute name)}, ')' ;

A compile-time error occurs if any of the following are not true:

- Each *relationship specification* in the *relationship formalization* must specify a valid navigation form the object being declared to a single instance of another object.
- Each *attribute name* must name an attribute of the object navigated to by the preceding *relationship specification*.
- Each attribute named by the *attribute name*, of the object navigated to by the preceding *relationship specification*, must have the same type as the attribute being declared.

In addition:

- All of the *referential attributes* of an object that formalize a specific relationship, i.e. the ones that have a *relationship specification* that navigates the same relationship, to the same object, must together form an identifier of the object being navigated to.
- A non-associative one-to-one relationship must be formalized in one and only one, of the participating objects.
- A non-associative one-to-many relationship must be formalized in the object on the many sided and nowhere else.
- A associative relationship must be formalized in the associative object and nowhere else.
- A associative object must be formalized in the associative relationship in both directions.

7. Relationships

Associations between objects in a domain provide information about the way instances of the objects are related to one another. Such associations are modeled as relationships between the respective objects.
Every relationship has a unique relationship number. At each end the relationship has:

- A role phrase.
- A multiplicity

The role phrase indicates the role played by the object attached to the end of the relationship near the role phrase.
The multiplicity specifies the range of allowable cardinalities that a set may assume. A multiplicity specification is shown as one or many.

7.1. Relationship Declarations

A *relationship declaration* declares a relationship between objects in a domain. The declaration specifies the relationship number of the relationship together with its definition.

[92] relationship declaration = 'relationship', relationship number, 'is', relationship definition, ';' ;

[93] relationship definition = subtype relationship definition | regular relationship definition ;

The definition of a relationship can either be a subtype relationship definition, in which case the relationship models a super/sub type hierarchy or a regular relationship definition, in which case the relationship models a regular relationship.
A compile-time error occurs if the relationship number naming a relationship is already declared as a relationship in the enclosing domain.

7.2. Regular Relationships

A *regular relationship definition* specifies a relationship between two objects in a domain, together with an optional *associative relationship* that specifies the object that formalizes the relationship.

[94] regular relationship definition = relationship description, ',', relationship description, [associative relationship] ;

[95] relationship description = object name, relationship conditionality, role name, relationship multiplicity, object name ;

[96] role name = identifier ;

[97] relationship conditionality = 'unconditionally' | 'conditionally' ;

[98] relationship multiplicity = 'one' | 'many' ;

[99] associative relationship = 'using', relationship multiplicity, object name ;

A *regular relationship definition* defines the relationship from the perspective of each of the participating objects. Therefore two relationships specifications are defined. A compile-time error occurs if any of the following are not true:

- Both of the object names in a *relationship description* must name an object that has already been declared, or pre-declared, in the enclosing domain.
- In a *regular relationship definition*, the first object in the first *relationship description* must be the same object as the second object in the second relationship description.
- In a *regular relationship definition*, the second object in the first *relationship description* must be the same object as the first object in the second relationship description.
- The object name in an *associative relationship* must name an object that has already been declared or pre-declared in the enclosing domain.
- The object in an *associative relationship* cannot have already been specified as an associative object in another relationship.
- The multiplicity of an *associative relationship* can only be one.

Notice that the multiplicity of each end of a relationship is not declared using an integer interval, but instead using a conditionality together with a multiplicity. Table 7.1, “Mappings for Integer Intervals to Conditionality and Multiplicity” gives the mapping between integer intervals and conditionality and multiplicity. Notice that this restricts the multiplicity that relationship ends can have.

Table 7.1. Mappings for Integer Intervals to Conditionality and Multiplicity

Integer Interval	Conditionality	Multiplicity
`1`	unconditionally	one
`0 .. 1`	conditionally	one
`1 .. *`	unconditionally	many

Integer Interval	Conditionality	Multiplicity
`0 .. *`	conditionally	many

Each relationship description is meant to read as a meaningful sentence.

7.2.1. Unconditional Relationships

There are three basic kinds of relationship; one-to-one, one-to-many and many-to-many. These basic relationships are unconditional because every instance of both objects must participate in the relationship.

7.2.1.1. One-to-one Relationships

A one-to-one relationship exists when a single instance of an object is associated with a single instance of another.

7.2.1.2. One-to-many Relationships

A one-to-many relationship exists when a single instance of an object is associated with one or more instances of another and each instance of the second object is associated with just one instance of the first.

7.2.1.3. Many-to-many Relationships

A many-to-many relationship exists when a single instance of an object is associated with one or more instances of another and each instance of the second object is associated with one or more instances of the first.

7.2.1.4. Reflexive Relationships

A reflexive relationship exists where instances of an object are related to instances of the same object.

7.2.2. Conditional Relationships

In unconditional relationships, every instance of the objects is required to participate in the relationship. In a conditional relationship there can be instances of the objects that do not participate. A conditional relationship is declared using the reserved word `conditionally` in the relationship conditionality.

7.3. Subtype Relationships

In many problems you find distinct specialized objects that have certain attributes in common. In this case, we abstract a more general object to represent the

characteristics shared by the original specialized objects. These objects related through a subtype relationship.

A *subtype relationship definition* defines a subtype relationship between an object and one or more other objects in a domain.

[100] subtype relationship definition = object name, 'is_a', '(', object name, {(',', object name}, ')' ;

A compile-time error occurs if the object names do not name objects that have already been declared or pre-declared in the enclosing domain.

The order of the subtype objects in a subtype relationship definition is arbitrary. An object can be the super-type in multiple sub-type relationships. An object can also be the subtype of multiple other objects.

7.3.1. Invalid Subtype Relationships

Certain groups of subtype relationship are invalid. It is important to remember that, if a subtype object has a subtype relationship, then all of its subtypes are subtypes of its super-type.

A subtype relationship is invalid and hence causes a compile-time error to occur if a super-type object is a subtype of one of its subtypes or a subtype object is a subtype of its super-type using more than one subtype relationship.

7.4. Relationship Specification

A *relationship specification* specifies which relationship is required to be created, navigated or deleted.

[101] relationship specification = relationship number, '.', role, '.', object name | relationship number, '.', role | relationship number, '.', object name | relationship number ;

[102] role = role name | 'is_a' ;

There are four forms of relationship specification. The analyst must ensure that the following are unambiguously specified:

- The relationship.
- The direction of the navigation.
- The destination object.

The relationship can be specified by:

- The relationship number.
- The relationship role, if it is unambiguous.

The direction of navigation can be specified by:

- Default (in non-reflexive relationships).
- Relationship role, if it is unambiguous.
- Object name, in non-reflexive relationships.

The destination object can be specified by:

- Default (in non-reflexive relationships).
- Relationship role.
- Object name.

A compile-time error occurs if insufficient information is provided to identify all the necessary parameters.

7.5. Correlated Relationship Specification

A correlated relationship navigation finds the instance or instances, of the associative object that is related to a pair of corresponding instances. A *correlated relationship specification* specifies which relationship is required to be correlatively navigated.

[103] correlated relationship specification = relationship number, '.', role name, '.', object name | relationship number, '.', role name | relationship number, '.', object name | relationship number ;

There are four forms of correlated relationship specification. The relationship must be unambiguously specified. If an object name is specified then it must be the name of the associative objects being navigated to. If a role name is specified then it must be the role at the destination end of the relationship for the first instance specified.

8. Lifecycles

Objects can be either passive or active. Active objects can have a lifecycle. States represents a stage in that lifecycle. Events cause an instance to change from one state to another. What new state is achieved when an instance of an object in a given state receives a particular event is defined within a state transition table.

Each associative object in a domain can also have an assigner lifecycle. Assigner lifecycles provide a single point of control through which competing requests are serialized. Because of this, there is only one copy of an assigner lifecycle for all instances of the associative object.

Assigner state represent a stage in that lifecycle. Assigner events cause an assigner to change from one state in its lifecycle to another. What new state is achieved when an assigner in a given state receives a particular event is defined within an assigner transition table.

8.1. States

A state declaration defines a state in the lifecycle of an object instance or an assigner object.

[104] state declaration = state modifier, 'state', state name, '(', parameter declaration list, ')', ';' ;

[105] state name = identifier ;

The state name in a state declaration may be used to refer to the state.

8.1.1. State Modifiers

[106] state modifier = 'assigner', ['start'] | 'creation' | 'terminal' ;

8.1.1.1. Assigner States

A state that is declared with the modifier `assigner` is called an assigner state. An assigner state is always entered without reference to a particular instance.

A compile-time error occurs if an attempt is made to reference the current instance using the reserved word `this` in the body of the state. A compile-time error also occurs if the enclosing object is not an associative object.

8.1.1.2. Start States

A state that is declared with the modifier `start` is called a start state. One and only one state for an assigner object must be declared as the start state. The start state is the state in which the assigner object starts its lifecycle.

A compile-time error occurs if an associative object has more than one assigner start state.

8.1.1.3. Creation States

A state that is declared with the modifier `creation` is called the *creation state*. A *creation state* is always entered without reference to a particular instance.

A compile-time error occurs if an attempt is made to reference the current instance using the reserved word `this` in the body of the state.

8.1.1.4. Instance States

A state that is declared without any modifiers is called an *instance state*. An *instance state* is always entered with respect to a particular instance, which becomes the instance to which the reserved word `this` refers during execution of the state body.

8.1.1.5. Terminal States

A state that is declared with the modifier `terminal` is called a terminal state. A *terminal state* does not necessarily imply that an instance in this state must cease to exist, it could simply stay in this state for historic purposes.

A compile-time error occurs if an attempt is made to delete the current instance using the reserved word `this` in the body of a non-terminal state.

8.1.2. Formal State Parameters

The optional formal parameters of a state, are specified by a list of comma-separated *parameter declarations*. Each *parameter declaration* consists of a name that specifies the name of the parameter, a parameter mode and a type. The following productions are repeated from Section 10.1.3, “Service Signature” to make the presentation clearer.

[131] parameter declaration list = parameter declaration, {(',', parameter declaration)} ;

[132] parameter declaration = parameter name, ':' parameter mode type ;

[133] parameter name = identifier ;

If a state has no parameters, only an empty pair of parentheses appears in the state's declaration.

A compile-time error occurs if two parameters are declared to have the same name.

When the state is entered, the values of the actual argument expressions initialize newly created parameter variables, each of the declared type, before execution of

the body of the state. The parameter name that appears in the parameter declaration may be used as a simple name in the body of the state to refer to the parameter.
The scope of parameter names is the entire body of the state.

8.1.2.1. Formal State Parameter Modes

The mode of a formal parameter conveys the direction of information transfer with the actual parameter. The following production is repeated from Section 10.1.3, “Service Signature” to make the presentation clearer.

[134] parameter mode = 'in' | 'out' ;
type

Formal parameters of mode `in` are passed from the event generation to the state. The state is not allowed to change the value of a formal parameter of mode `in`.
A compile-time error occurs if a formal parameter for a state is declared as mode `out`.

8.1.3. State Signature

The signature of a state consists of the name of the state and the number and types of parameters to the state.
A compile-time error occurs if an object declares two states with the same signature.

8.1.4. State Definition

A *state definition* provides the specification of the behaviour of the a state.

[107] state definition = state modifier, 'state', domain name, '::', object name, '.', state name, '(', parameter declaration list, ')', 'is', {variable declaration, 'begin', handled sequence of statements, 'end', ['state'], ';' ;

A compile-time error occurs of the signature of the state, defined by the state name together with the parameter declaration list has not already been declared by a state declaration within the object named, by the object name. In addition, a compile-time error occurs if the state modifiers do not match the state modifiers of the corresponding state declaration.

8.2. Events

An event is the abstraction of an incident or signal in the real world, that signals that an object instance is moving to a new state.

[108] event declaration = event modifier, 'event', event name, '(', parameter declaration list, ')', ';' ;

[109] event name = identifier ;

The event name in an *event declaration* may be used to refer to the event.

8.2.1. Event Modifiers

[110] event modifier = 'assigner' | 'creation' ;

8.2.1.1. Assigner Events

An event that is declared with the modifier `assigner` is called an assigner event. An assigner event is always generated without reference to a particular instance. A compile-time error occurs if the enclosing object is not an associative object.

8.2.1.2. Creation Events

An event that is declared with the modifier `creation` is called a creation event. A *creation event* is always generated without reference to a particular instance.

8.2.1.3. Instance Events

An event that is declared without any modifiers is called an *instance event*. An *instance event* is always generated to a particular instance.

8.2.1.4. Polymorphic Events

All the instance events declared by a super-type object are *polymorphic events*. *Polymorphic events* can be used in the non-assigner transition tables of all the subtype objects of a super-type object.

8.2.2. Formal Event Parameters

The optional *formal parameters* of an event are specified by a list of comma-separated parameter declarations. Each parameter declaration consists of a name that specifies the name of the parameter, a parameter mode and a type. The following productions are repeated from Section 10.1.3, “Service Signature” to make the presentation clearer.

[131] parameter declaration list = parameter declaration, {(',', parameter declaration)} ;

[132] parameter declaration = parameter name, ':' parameter mode type ;

[133] parameter name = identifier ;

If an event has no parameters, only an empty pair of parentheses appears in the event declaration.
A compile-time error occurs if two parameters are declared to have the same name.
When an event is generated (see Section 12.23, “The Generate Statement”), the values of the actual argument expressions initialize newly created parameter variables, each of the declared type.

8.2.2.1. Formal Event Parameter Modes

The mode of a formal parameter conveys the direction of information transfer with the actual parameter. The following productions is repeated from Section 10.1.3, “Service Signature” to make the presentation clearer.

[134] parameter mode = 'in' | 'out' ;
type

Formal parameters of mode `in` are passed from the event generation to the state. The state is not allow to change the value of a formal parameter of mode `in`.
A compile-time error occurs if a formal parameter for an event is declared as mode `out`.

8.2.3. Event Signature

The *signature* of an event consists of the name of the event and the name and types of parameters to the event.
A compile-time error occurs of an object declares two events with the same signature.

8.3. Transitions

A transition table declaration defines the transitions in the lifecycle of an object instance or an assigner object.

[111] transition table declaration = transition modifier, 'transition', 'is', transition row, {transition row}, 'end', ['transition'], ';' ;

8.3.1. Transition Modifiers

[112] transition modifier = 'assigner' ;

8.3.1.1. Assigner Transitions

A transition table that is declared with the modifier `assigner` specifies the assigner lifecycle for an object.

A compile-time error occurs if the enclosing object is not an associative object. In addition, a compile-time error occurs if an associative object declares more than one assigner transition table.

8.3.1.2. Instance Transitions

A transition table that is not declared with the modifier `assigner` specifies the instance lifecycle for an object.

A compile-time error occurs if the object declares more than one instance transition table.

8.3.2. Transition Rows

Each *transition row* in a transition table declaration specifies the resultant state achieved when an event is received by either:

- An instance in a given state.
- An assigner object in a given state.
- An object, for a creation events.

[113] transition row = initial state, '(', transition, {(',', transition)}, ';' ;

[114] initial state = 'Non_Existent' | state name) ;

[115] transition = incoming event, '=>', resultant state ;

[116] incoming event = object name, '.', event name | event name ;

[117] resultant state = 'Ignore' | 'Cannot_Happen' | state name ;

For a non-assigner state transition table, a compile-time error occurs if the following is not true for each transition row:

- The initial state must name either an instance, creation or terminal state of the enclosing object or be the reserved word `Non_Existent`.
- For each transition within a transition row,
  - The incoming event must name either an instance or creation event of the enclosing object or a polymorphic event from a super-type of the enclosing object.
  - The resultant state must name either an instance or terminal state of the enclosing object or be one of the reserved words `Ignore` or `Cannot_Happen`.

    - If the resultant state is either an instance or terminal state, then the parameter declaration list of the incoming event must match the parameter declaration list of the resultant state.

- There must be one and only one, transition for every instance or creation events of the enclosing object.

- There must be one and only one, transition for every polymorphic event of every super-type of the enclosing object.

- If the initial state of a transition row is the reserved word `Non_Existent`, then all of the transitions with creation events must result in a creation state. In addition, all of the the transitions with instance or polymorphic event must result in the reserved word `Cannot_Happen`.

- If the initial state of a transition row is the reserved word `Non_Existent`, then all of the transitions with instance or polymorphic events must result in the reserved word `Cannot_Happen`.

- If the initial state of a transition row is either an instance or creation state, then all of the transitions with creation events must result in the reserved word `Cannot_Happen`.

- If the initial state of a transition row is a terminal state, then all of the transitions must result in either of the reserved words `Ignore` or `Cannot_Happen`.

In addition a compiler error also occurs if the following is not true:

- There must be one and only one, transition row for every instance, creation and terminal state of the enclosing object.

- There must be one and only one, transition row for the reserved word `Non_Existent`.

Similarly, for an assigner transition table, a compile-time error occurs if the following is not true for each transition row:

- The initial state must name an assigner state of the enclosing object.

- For each transition with a transition row:

    - The incoming event must name an assigner event of the enclosing object.

  - The resultant state must name either an assigner state of the enclosing object or be one of the reserved words `Ignore` or `Cannot_Happen`.
  - If the resultant state is an assigner state, then the parameter declaration list of the incoming event must match the parameter declaration list of the resultant state.
- There must be one and only one transition for every assigner state of the enclosing object.

Additionally, for an assigner state transition table, a compile-time error occurs if there is not one and only one transition row for every assigner state of the enclosing object.

The meaning of the reserved words `Cannot_Happen`, `Ignore` and `Non_Existent` is described in Table 8.1, “Reserved States”.

Table 8.1. Reserved States

Name	Description
`Cannot_Happen`	Specifies that an exception is raised to indicate that an illegal event has happened.
`Ignore`	Specifies the event is ignored.
`Non_Existent`	Specifies for a non-assigner state transition table that the instance does not yet exist.

9. Functions

A function is a unit of executable code that can be invoked, passing a fixed number of values as arguments and returns a value of a given type.
There are two forms of functions; domain functions and object functions.

9.1. Domain Functions

A domain function allows a domain to be thought of as a black box. Domain functions provide an interface to entities outside the domain.
A *domain function declaration* declares a domain function.

[118] domain function declaration = domain function modifier, 'function', function name, '(', function parameter declaration list, ')', 'return', type, [raises expression], ';' ;

[119] function name = identifier ;

9.1.1. Domain Function Modifiers

[120] domain function modifier = 'public' | 'private' ;

9.1.1.1. Public Domain Functions

If a domain function is declared with the modifier `public`, then it may be accessed by any code that can access the domain in which it is declared.

9.1.1.2. Private Domain Functions

If a domain is declared with the modifier `private`, then access is permitted only when it occurs from within the domain in which it is declared.

9.1.2. Formal Parameters

The formal parameter of a domain function, if any, are specified by a list of comma-separated parameter declarations. Each parameter declaration consists of a name that specifies the name of the parameter, a parameter mode and a type.

[121] function parameter declaration list = function parameter declaration, {(',', function parameter declaration)} ;

[122] function parameter declaration = function parameter name, ':', 'in', type ;

[123] function = identifier ;
parameter name

If a function has no parameters, only an empty pair of parentheses appears in the function declaration.

A compile-time error occurs if any two parameters are declared to have the same name.

When the function is invoked (see Section 14.4, “Primary Expressions”), the values of the actual argument expressions initialize newly created parameter variables, each of the declared type, before the execution of the body of the function. The parameter name that appears in the parameter declaration may be used as a simple name in the body of the function to refer to the parameter. The scope of the parameter names is the entire body of the function. The body the function must return a value of the declared function return type.

9.1.3. Function Signatures

The signature of a function consists of the name of the function, the name and types of parameters to the function and the type of the parameter returned from the function.

A compile-time error occurs if a domain declares two domain functions with the same signature.

9.1.4. Overloading

If two domain functions of a domain have the same name but different signatures, then the function is said to be overloaded. When a domain function is invoked (see Section 14.4, “Primary Expressions”), the number of arguments and the compile-time types of the arguments are used to determine the signature of the function that will be invoked.

A function cannot be overloaded on the return type. A compile time error will be raised if two functions have the same name, same input parameter profile but different return types.

9.1.5. Raises Expression

A raises expression is used to declare any exceptions (see Chapter 11, *Exceptions* that can result from the execution of a function.

[135] raises = 'raises', '(', exception specification, {exception
expression specification}, ')' ;

The following is repeated from Section 11.4, “Handling of an Exception” to make the representation clearer:

[149] exception specification = domain name, '::', exception name | exception name ;

A compile-time error occurs if an exception that can result from the execution of the body of a function is not mentioned in the raises expression in the declaration of the function.

See Chapter 11, *Exceptions* for more information on exceptions.

9.1.6. Domain Function Definition

A domain function definition provides the specification of the behavior of a domain function.

[124] domain function definition = 'function', domain name, '::', function name, '(', function parameter declaration list, ')', 'return', type, [raises expression], 'is', {variable declaration}, 'begin' handled sequence of statements, 'end', ['function'], ';' ;

A compile-time error occurs if:

- The signature of the function, named by the function name together with the parameter declaration list has not already been declared, by a domain function declaration, as a domain function of the domain named by the domain name.
- The modifiers do not match the modifiers of the corresponding domain function declaration.
- The raise expression does not match the raises expression of the corresponding domain function declaration.
- The *handled sequence of statements* does not include at least one `return` statement.

9.2. Object Functions

[125] object function declaration = object function modifier, 'function', function name, '(', function parameter declaration list, ')', 'return', type, [raises expression], ';' ;

9.2.1. Object Function Modifiers

[126] object function modifier = ('public' | 'private'), ['instance', ['deferred', '(', relationship number, ')']] ;

9.2.1.1. Public Object Functions

If an object function is declared with the modifier `public`, then it may be accessed by any code that can access the object in which it is declared.

By default, all object functions are declared `public`.

9.2.1.2. Private Object Functions

If an object function is declared with the modifier `private`, then access is permitted only when it occurs from with the object in which it is declared.

9.2.1.3. Instance Object Functions

An object function that is declared with the modifier `instance` is called an instance function. An instance function is always invoked with respect to a particular instance, which becomes the current instance to which the reserved word `this` refers, during execution of the function body. An object function that is not declared with the modifier `instance` is called a non-instance function. A non-instance function is always invoked without reference to a particular instance. A compile-time error occurs if an attempt is made to reference the current instance using the reserved word `this` in the body of a non-instance function.

9.2.1.4. Deferred Object Functions

A object function that is declared with the modifier `deferred` is called a deferred function. Only instance functions can be deferred.

A deferred function is an instance function whose implementation is deferred to a subtype of the enclosing object. Which subtype hierarchy the function is deferred to is given by the relationship number after the reserved word `deferred`

A compile-time error occurs if the following are not true:

- The relationship number specified after the reserved word `deferred` must be a subtype relationship of the enclosing object.
- For every direct subtype in the subtype hierarchy that the function is deferred to, the subtype must declare an instance function with the same signature as the deferred function.

An instance function that has been deferred to an object by one of its super-types can also be deferred, by the object, to one of its subtype hierarchies.

When the function is invoked, which implementation is actually executed depends upon the current subtype of the instance that the function was invoked on.

A deferred function is always invoked with respect to a particular instance, the current subtype instance of this instance becomes the current instance to which the reserved word `this` refers during execution of the appropriate function body.

9.2.2. Formal Parameters

The formal parameters of an object function, if any, are specified by a list of comma-separated parameter declarations. Each parameter declaration consists of a name that specifies the name of the parameter, a parameter mode and a type.

[121] function parameter declaration list = function parameter declaration, {(',', function parameter declaration)} ;

[122] function parameter declaration = function parameter name, ':', 'in', type ;

[123] function parameter name = identifier ;

If a function has no parameters, only an empty pair of parentheses appears in the function declaration.

A compile-time error occurs if any two parameters are declared to have the same name.

When the function is invoked (see Section 14.4, “Primary Expressions”), the values of the actual argument expressions initialize newly created parameter variables, each of the declared type, before execution of the body of the function. The parameter name that appears in the parameter declaration may be used as a simple name in the body of the function to refer to the parameter.

The scope of parameter names is the entire body of the function. The body of the function must return a value of the declared function type.

9.2.3. Function Signatures

The signature of a function consists of the name of the function, the name and types of parameters to the function and the type of the parameter returned from the function.

A compile-time error occurs if an object declares two domain functions with the same signature.

9.2.4. Overloading

If two object functions of an object have the same name but different signatures, then the function is said to be overloaded. When a domain function is invoked (see

Section 14.4, “Primary Expressions”), the number of arguments and the compile-time types of the arguments are used to determine the signature of the function that will be invoked.

A function cannot be overloaded on the return type. A compile time error will be raised if two functions have the same name, same input parameter profile but different return types.

9.2.5. Raises Expression

A raises expression is used to declare any exceptions (see Chapter 11, *Exceptions* that can result from the execution of a function.

[135] raises expression = 'raises', '(', exception specification, {exception specification}, ')' ;

The following is repeated from Section 11.4, “Handling of an Exception” to make the representation clearer:

[149] exception specification = domain name, '::', exception name | exception name ;

A compile-time error occurs if an exception that can result from the execution of the body of a function is not mentioned in the raises expression in the declaration of the function.

See Chapter 11, *Exceptions* for more information on exceptions.

9.2.6. Object Function Definition

An *object function definition* provides the specification of the behaviour of an object service.

[127] object function definition = object function modifier, 'function', domain name, '::', object name, '.' function name, '(', function parameter declaration list, ')', 'return', type, [raise expression], 'is', {variable declaration}, 'begin', handled sequence of statements, 'end', ['function'], ';' ;

A compile-time error occurs if:

- The signature of the function, named by the function name together with the *parameter declaration list* has not already been declared, by an *object function definition*, as a function of the object named by the object name, in the domain named by the domain name.
- The modifiers do not match the modifiers of the corresponding object function declaration.

- The raise expression does not match the raises expression of the corresponding domain function declaration.
- The *handled sequence of statements* does not include at least one `return` statement.

10. Services

A service is a unit of executable code that can be invoked, passing a fixed number of values as arguments.

There are two forms of services; domain services and object services.

10.1. Domain Services

A domain service allows a domain to be thought of as a black box. Domain services provide an interface to entities outside of the domain.

A *domain service declaration* declares a domain service.

[128] domain service declaration = domain service modifier, 'service', service name, '(', parameter declaration list, ')', [raises expression], ';' ;

[129] service name = identifier ;

The service name in a *domain service declaration* may be used to refer to the service.

10.1.1. Domain Service Modifiers

[130] domain service modifier = 'public' | 'private' | 'native' ;

10.1.1.1. Public Domain Services

If a domain service is declared with the modifier `public`, then it may be accessed by any code that can access the domain in which it is declared. By default, all domain services are declared public.

10.1.1.2. Private Domain Services

If a domain service is declared with the modifier `private`, then access is permitted only when it occurs from within the domain in which it is declared.

10.1.1.3. Native Domain Services

If a domain service is declared with the modifier `native`, then it may be accessed by an code that can access the domain in which it is declared. The body of the service is implemented in a native language.

10.1.2. Formal Parameters

The `formal parameters` of a domain service, if any, are specified by a list of comma-separated *parameter declarations*. Each *parameter declaration* consists of a name the specifies the name of the parameter, a parameter mode and type.

[131] parameter declaration list = parameter declaration, {(',', parameter declaration)} ;

[132] parameter declaration = parameter name, ':' parameter mode type ;

[133] parameter name = identifier ;

If a service has no parameters, only an empty pair of parentheses appears in the services declaration.

A compile-time occurs if any two parameters are declared to have the same name, or if a parameter of mode out is declared as an unconstrained array type.

When the service is invoked (see Section 12.9, “Invocation Statements”, the values of the actual argument expressions initialize newly created parameter variables, each of the declared type, before execution of the body of the service. The parameter name that appears in the parameter declaration may be used as a simple name in the body of the service to refer to the parameter.

The scope of the parameter names is the entire body of the service.

10.1.2.1. Formal Parameter Modes

The mode of a formal parameter conveys the direction of information transfer of the actual parameter.

[134] parameter mode type = 'in' | 'out' ;

Formal parameters of mode `in` are passed from the caller to the service. The service is not allowed to change the value of a formal parameter of mode `in`.

Formal parameters of mode `out` are passed from the service back to the caller on completion of the service. This means that the caller must supply an expression that denotes a variable as the value of the actual argument expression. On completion of the service the value of the formal parameter is assigned to the supplied variable.

10.1.3. Service Signature

The signature of a domain service consists of the name of the service and the name and types of parameters to the service.

A compile-time error occurs if a domain declares two domain services with the same signature.

10.1.4. Overloading

If two services of a domain have the same but different signatures, then the service is said to be overloaded. When a domain service is invoked (see Section 12.9, “Invocation Statements”), the number of arguments and the compile-time types of the arguments are used to determine the signature of the service that will be invoked.

10.1.5. Raises Expression

A *raises expression* is used to declare any exceptions (see Section 11.3, “Compile-Time Checking of Exceptions”) that can result from the execution of a service.

[135] raises expression = 'raises', '(', exception specification, {exception specification}, ')' ;

The following production is repeated from Section 11.4, “Handling of an Exception” to make the presentation here clearer.

[149] exception specification = domain name, '::', exception name | exception name ;

A compile-time error occurs if an exception that can result from the execution of the body of a service is not mentioned in the raises expression in the declaration of the service.

See Chapter 11, *Exceptions* for more information about exceptions.

10.1.6. Domain Service Definition

A *domain service definition* provides the specification of the behaviour of a domain service.

[136] domain service definition = ('native', 'service', domain name, '::', service name, '(', parameter declaration list, ')', [raises expression], 'is', native code) | (domain service modifier, 'service', domain name, '::', service name, '(', parameter declaration list, ')', [raise expression], 'is', {variable declaration}, 'begin', handled sequence of statements, 'end', ['service'], ';' ;

[137] native code = '$NATIVE', line terminator, {native code line}, '$ENDNATIVE', line terminator ;

[138] native code line = {(letter | digit | other character | underscore | white space)}, line terminator ;

The native code is a sequence of code statements embraced by `$NATIVE` and `$ENDNATIVE`. All statements between these two markers are ignored for the purposes of MASL parsing.

A compile-time error occurs if:

- The signature of the service, named by the service name together with the parameter declaration list has not already been declared by a domain service declaration.
- The modifiers do not match the modifiers of the corresponding domain service declaration.
- The raises expression does not match the raise expression of the corresponding domain service declaration.

10.2. Object Services

An object service gives an interface to an object. An *object service declaration* declares an object service.

[139] object service declaration = object service modifier, 'service', service name, '(', parameter declaration list, ')', [raise expression], ';' ;

The service name in an object service declaration may be used to refer to the service.

10.2.1. Object Service Modifiers

[140] object service modifier = ('public' | 'private' | 'native'), ['instance', [('deferred', '(', relationship number, ')')]] ;

10.2.1.1. Public Object Services

If an object service is declared with the modifier `public`, then it may be accessed by any code that can access the object in which it is declared.

By default, all object services are declared `public`.

10.2.1.2. Private Object Services

If an object service is declared with the modifier `private`, then access is permitted only when it occurs from within the object in which it is declared.

10.2.1.3. Instance Object Services

An object service that is declared with the modifier `instance` is called an instance service An instance service is always invoked with respect to a particular instance, which becomes the current instance to which the reserved word `this` refers, during execution of the service body. An object service that is not declared with the modifier `instance` is called a non-instance service A non-instance service is always invoked without reference to a particular instance. A compile-time error occurs if an attempt is made to reference the current instance using the reserved word `this` in the body of a non-instance service.

10.2.1.4. Deferred Object Services

A object service that is declared with the modifier `deferred` is called a deferred service. Only instance services can be deferred.

A deferred service is an instance service whose implementation is deferred to a subtype of the enclosing object. Which subtype hierarchy the service is deferred to is given by the relationship number after the reserved word `deferred`.

A compile-time error occurs if the following are not true:

- The relationship number specified after the reserved word `deferred` must be a subtype relationship of the enclosing object.
- For every direct subtype in the subtype hierarchy that the service is deferred to, the subtype must declare an instance service with the same signature as the deferred service.

An instance service that has been deferred to an object by one of its super-types can also be deferred, by the object, to one of its subtype hierarchies.

When the service is invoked, which implementation is actually executed depends upon the current subtype of the instance that the service was invoked on.

A deferred service is always invoked with respect to a particular instance, the current subtype instance of this instance becomes the current instance to which the reserved word `this` refers during execution of the appropriate service body.

10.2.2. Formal Parameters

The `formal parameters` of an object service, if any, are specified by a list of comma-separated *parameter declarations*. Each *parameter declaration* consists of a name the specifies the name of the parameter, a parameter mode and type.

[131] parameter = parameter declaration, {(',', parameter declaration)} ; declaration list

[132] parameter declaration = parameter name, ':' parameter mode type ;

[133] parameter name = identifier ;

If a service has no parameters, only an empty pair of parentheses appears in the services declaration.
A compile-time error occurs if any two parameters are declared to have the same name, or if a parameter of mode out is declared as an unconstrained array type.
When the service is invoked (see Section 12.9, “Invocation Statements”, the values of the actual argument expressions initialize newly created parameter variables, each of the declared type, before execution of the body of the service. The parameter name that appears in the parameter declaration may be used as a simple name in the body of the service to refer to the parameter.
The scope of the parameter names is the entire body of the service.

10.2.2.1. Formal Parameter Modes

The mode of a formal parameter conveys the direction of information transfer of the actual parameter.

[134] parameter mode type = 'in' | 'out' ;

Formal parameters of mode `in` are passed from the caller to the service. The service is not allowed to change the value of a formal parameter of mode `in`.
Formal parameters of mode `out` are passed from the service back to the caller on completion of the service. This means that the caller must supply an expression that denotes a variable as the value of the actual argument expression. On completion of the service the value of the formal parameter is assigned to the supplied variable.

10.2.3. Service Signature

The signature of a object service consists of the name of the service and the name and types of parameters to the service.
A compile-time error occurs if a object declares two domain services with the same signature.

10.2.4. Overloading

If two services of a object have the same but different signatures, then the service is said to be overloaded. When an object service is invoked (see Section 12.9, “Invocation Statements”), the number of arguments and the compile-time types of the arguments are used to determine the signature of the service that will be invoked.

10.2.5. Raises Expression

A *raises expression* is used to declare any exceptions (see Section 11.3, “Compile-Time Checking of Exceptions”) that can result from the execution of a service.

[135] raises expression = 'raises', '(', exception specification, {exception specification}, ')' ;

The following production is repeated from Section 11.4, “Handling of an Exception” to make the presentation here clearer.

[149] exception specification = domain name, '::', exception name | exception name ;

A compile-time error occurs if an exception that can result from the execution of the body of a service is not mentioned in the raises expression in the declaration of the service.

See Chapter 11, *Exceptions* for more information about exceptions.

10.2.6. Object Service Definition

An *object service definition* provides the specification of the behaviour of a object service.

[141] object service definition = ('native', 'service', domain name, '::', object name, '.' service name, '(', parameter declaration list, ')', [raise expression], 'is', native code) | (object service modifier, 'service', domain name, '::', object name, '.' service name, '(', parameter declaration list, ')', [raises expression], 'is', {variable declaration}, 'begin', handled sequence of statements, 'end', ['service'], ';' ;

[137] native code = '$NATIVE', line terminator, {native code line}, '$ENDNATIVE', line terminator ;

[138] native code line = {(letter | digit | other character | underscore | white space)}, line terminator ;

The native code is a sequence of code statements embraced by `$NATIVE` and `$ENDNATIVE`. All statements between these two markers are ignored for the purposes of MASL parsing.

A compile-time error occurs if:

- The signature of the service, named by the service name together with the parameter declaration list has not already been declared by an object service declaration.

- The modifiers do not match the modifiers of the corresponding object service declaration.
- The raises expression does not match the raise expression of the corresponding object service declaration.

11. Exceptions

When a program violates the semantic constraints of the language, an error is signaled as an *exception*. An example of such a violation is an attempt to index outside the bounds of an array. This causes a non-local transfer of control from the point where the *exception* occurred to a point that can be specified by the analyst. An *exception* is said to be *raised* from the point where it occurred and is said to be *caught* at the point to which control is transferred. Analysts can also raise exceptions explicitly, using `raise` statements.

Every exception is represented by an exception type. Handlers are established by exception handlers, which consist of handled sequence of statements. During the process of raising an exception, the software architecture abruptly completes one by one, any expressions, statements invocations and initializers that have begun but not completed execution in the current thread. This process continues until a handler is found that indicates that it handles that particular exception by naming the exception type. If no such handler is found, then the software architecture is notified.

This chapter describes the different causes of exceptions. It details how exceptions are checked at compile-time and processed at run-time.

11.1. Exception Declarations

An *exception declaration* declares a new exception type. The declaration specifies the name of the exception.

[142] exception declaration = exception modifier, 'exception', exception name, ';' ;

[143] exception name = identifier ;

11.1.1. Exception Modifiers

[144] exception modifier = 'public' | 'private' ;

11.1.1.1. Public Exceptions

If an exception is declared with the modifier `public`, then it may be accessed by any code that can access the domain in which it is declared. By default, all exceptions are declared `public`.

11.1.1.2. Private Exceptions

If an exception is declared with the modifier `private`, then access is only permitted from within the domain in which it is declared.

11.2. The Causes of Exceptions

An exception is thrown for one of two reasons.

- An abnormal execution condition was synchronously detected by the software architecture. Such conditions arise because:
  - Evaluation of an expression violates the normal semantics of the language, such as an integer divided by zero.
  - Some limitation of a resource is exceeded, such as using too much memory.

  These exceptions are not raised at an arbitrary point in the program, but rather at a point where they are specified or as possible result of an expression evaluation or statement execution.
- A `raise` statement was executed.

11.3. Compile-Time Checking of Exceptions

The language checks at compile-time, that a program contains handlers for exception types, by analyzing which exceptions can result from execution of a service. For each exception type, which is a possible result, the raises expression for the service must mention the exception type. The standard language defines a number of exception types. An analyst may declare additional exception types.

11.4. Handling of an Exception

When an exception is raised, control is transferred from the code that caused the exception to the nearest dynamically-enclosing handler of a handled sequence of statements that handles the exception.

[145]	handled sequence of statements	=	sequence of statements, [('exception', {exception handler}, [exception others handler]] ;
[146]	exception handler	=	'when', exception choice, '=>', sequence of statements ;
[147]	exception others handler	=	'when', 'others', '=>', sequence of statements ;
[148]	exception choice	=	exception specification ;
[149]	exception specification	=	domain name, '::', exception name | exception name ;

A statement or expression is dynamically enclosed by an exception handler if either of the following are true:

- It appears within the sequence of statements of which the exception handler is a part.
- If the caller of the statement or expression is dynamically enclosed by the exception handler.
- The caller of a statement or expression is the service invocation statement that was executed to cause the service to be invoked.

Whether a particular exception handler handles an exception is determined by comparing the name of the exception to the name of the exception given in the exception handler.

At most one `others` exception is allowed at the end of the exception handler list. This handler handles any exception that has not already been handled in the subsequent handlers within the list.

The control transfer that occurs when an exception is thrown causes abrupt completion of expressions and statements until an exception handler is encountered that can handle the exception. Execution then continues by executing the sequence of statements of that exception handler. The code that caused the exception is never resumed.

If no exception handler can be found, then the current thread (the thread that encountered the exception) is terminated, but only after the software architecture is notified. The exact rules for abrupt completion and for the catching of exceptions are specified in detail within the specification of each statement (see Chapter 12, *Statements*).

11.4.1. Handling Exceptions Across Asynchronous Boundaries

There are certain statements that cause asynchronous control. These are:

- Event generation.
- Asynchronous service invocation of a domain service of another domain.

If an exception is raised as a result of consuming an event and the exception is not handled, then the exception is not propagated back to the original event generation. Similarly, for the asynchronous invocation of a domain service. This is the opposite of synchronous invocation of a domain or object service, where the exception is propagated back to the caller. Hence, if an exception is raised in a state action and

this exception is not caught within this state action then the exception is passed to the software architecture.

12. Statements

A statement defines an action to be performed upon its execution. Some statements contain other statements as part of their structure; such other statements are sub-statements of the statement. In the same manner, some statements contain expressions as part of their structure.

The first section of this chapter discusses the distinction between normal and abrupt completion of statements. The remaining sections explain the various kinds of statements, describing in detail both their normal behaviour and any special treatment of abrupt completion.

12.1. Normal and Abrupt Completion of Statements

Every statement has a normal mode of execution in which certain computational steps are carried out. The following sections describe the normal mode of execution for each kind of statement. If all the steps are carried out as described, with no indication of abrupt completion, the statement is said to complete normally. However, certain events may prevent a statement from completing normally:

- The `exit` statement causes a transfer of control that may prevent normal completion of statements that contain them.
- Evaluation of certain expressions may raise exceptions; these exceptions are summarized in Section 14.3, “Normal and Abrupt Completion of Evaluation”. An explicit `raise` statement also results in an exception. An exception causes a transfer of control that may prevent normal completion of statements.

If such an event occurs, then execution of one or more statements may be terminated before all steps of their normal mode of execution have completed; such statements are said to complete abruptly.

The terms complete normally and complete abruptly also apply to the evaluation of expressions. The only reason an expression can complete abruptly is that an exception is raised because of a run-time exception or error.

If statement evaluates an expression, abrupt completion of the expression always causes the immediate abrupt completion of the statement, with the same reason. All succeeding steps in the normal mode of execution are not performed.

Unless otherwise specified in this chapter, abrupt completion of a expression causes the immediate abrupt completion of the statement itself, with the same reason and all succeeding steps in the normal mode of execution of the statement are not performed.

Unless otherwise specified in this chapter, abrupt completion of a sub-statement cause the immediate abrupt completion of the statement itself, with the same reason

and all succeeding steps in the normal mode of execution of the statement are not performed.

Unless otherwise specified, a statement completes normally if all expressions it evaluates and all sub-statements it executes, complete normally.

12.2. Sequence of Statements

A *sequence of statements* is a sequence of one or more statements.

[150] sequence of statements = statement, {statement} ;

A *sequence of statements* is executed by executing each of the statements in order from first to last (left to right). If all of these statements complete normally, then the sequence of statements completes normally. If any of these statements complete abruptly for any reason, then the *sequence of statements* completes abruptly for the same reasons.

12.3. Statements

There are many kinds of statements in the language. A *statement* is either simple or compound. A *simple statement* encloses no other statement. A *compound statement* encloses other statements as part of their structure.

[151] statement = simple statement | compound statement ;

[152] simple statement = null statement | assignment statement | output stream | input stream | input line stream | invocation statement | exit statement | raise statement | delay statement | link statement | unlink statement | associate statement | unassociate statement | generate statement | pragma statement ;

[153] compound statement = block statement | if statement | case statement | for statement | while statement ;

12.4. The Null Statement

The *null statement* does nothing.

[154] null statement = 'null', ';' ;

Execution of a null statement always completes normally.

12.5. Assignment

An assignment statement assigns the value of the right-hand side to the entity denoted by the left-hand side.

[155] assignment statement = {name}, ':=', extended expression, ';' ;

A compile-time error occurs if either of the following are true:

- The type of the right-hand side is not assignable to the type of left-hand side.
- The entity denoted by the left-hand side is not assignable (see Section 13.5, “Assignability of Names” for a description of names that are assignable and names that are not).

If run-time evaluation of either the left-hand side or the right-hand side completes abruptly, then the assignment statement completes abruptly for the same reason and no assignment occurs.

12.6. Output Stream

The output stream statement outputs the value of the right-hand side to the device denoted by the left-hand side. This can be a *name* or *device literal*. The only *device literal* available is `console` which maps to the standard output. The iteration allows multiple expressions to be output to the device in a single *simple statement*.

[156] output statement = (name, '<<', expression, {('<<', expression}, ';') | device literal, '<<', expression, {('<<', expression}, ';') ;

A compile-time error occurs if the type of *name* on the left-hand side is not of type device.

12.7. Input Stream

The *input stream* statement inputs the value from input of the left-hand side device to the right-hand side. The left-hand side can be a *name* or *device literal*. The only *device literal* available is `console` available is `console` which maps to the standard input. The iteration allows multiple expressions to be input from the device in a single *simple statement*.

[157] input stream = (name, '>>', name, {('>>', name}, ';') | device literal, '>>', name, {('>>', name}, ';') ;

A compile-time error occurs if the entity denoted by the right-hand side is not assignable (see Section 13.5, “Assignability of Names” for a description of names that are assignable and names that are not).

12.8. Input Line Stream

The *input line stream* statement inputs the value from input of the left-hand side device to the right-hand side. The value is a stream of data terminated by a new line character. The left-hand side can be a *name* or *device literal*. The only *device literal* available is `console` which maps to the standard input. The iteration allows multiple expressions to be input from the device in a single *simple statement*.

[158] input line stream = (name, '>>>', name, {('>>>', name}, ';') | device literal, '>>>', name, {('>>>', name}, ';') ;

A compile-time error occurs if the entity denoted by the right-hand side is not assignable (see Section 13.5, “Assignability of Names” for a description of names that are assignable and names that are not).

12.9. Invocation Statements

An *invocation statement* is used to invoke a domain, object or instance service.

[159] invocation statement = domain service invocation | object service invocation | instance service invocation ;

12.9.1. Domain Service Invocation

A *domain service invocation* is used to invoke a domain service.

[160] domain service invocation = (domain name, '::', service name, '(', argument list, ')', ';') | (service name, '(', argument list, ')', ';') ;

A compile-time error occurs if either of the following are true:

- The signature of the domain service to be invoked, named by the *service name* together with the *argument list* has not been declared, by a *domain service declaration*, as a service of either:
  - The domain named by the *domain name*, in the form domain_name`::service_name`.
  - The enclosing domain, in the form consisting of just a `service name`.
- The *domain service declaration* is not accessible (see Section 12.9.4, “Compile-Time Processing of Service Invocations”) to the invocation statement.

A *domain service invocation* is executed by evaluating the argument expressions. If the evaluation of any argument expression completes abruptly, then the invocation

statement completes abruptly for the same reason. The domain service definition is then executed. If this completes normally any necessary assigning back of formal to actual parameters occurs.

12.9.2. Object Service Invocation

An *object service invocation* is used to invoke an object based service.

[161] object service invocation = (object name, '.', service name, '(', argument list, ')', ';') | (instance expression, '.', service name, '(', argument list, ')', ';') | (service name, '(', argument list, ')', ';') ;

A compile-time error will occur if:

- The signature of the object base service to be invoked, named by the service name together with the argument list has not been declared, by an object service declaration, as a non-instance service of either:
  - The object, in the enclosing domain, named by the object name, in the form `object_name.service_name`.
  - The object, given by the object of the instance type of the instance expression, in the form *instance_expression*`.service_name`.
  - The enclosing object, in the form consisting of just a `service name`.
- The object service declaration is not accessible (see Section 12.9.4, “Compile-Time Processing of Service Invocations”) to the invocation statement.

An *object service invocation* is executed by evaluating the argument expressions. If the evaluation of any argument expression completes abruptly, then the object service invocation completes abruptly for the same reason. Note, that for an object service invocation of the form `instance_expression.service_name`, the instance expression is not evaluated. The object service definition is then executed. If this completes normally, any necessary assigning back of formal to actual parameters occurs.

12.9.3. Instance Service Invocation

An *instance service invocation* is used to invoke an object instance service.

[162] instance service invocation = (instance expression, '.', service name, '(', argument list, ')', ';') | (service name, '(', argument list, ')', ';') ;

A compile-time error occurs if:

- The signature of the object instance service to be invoked, named by the service name together with the argument list has not been declared, by an object service declaration, as a non-instance service of either:
  - The object, given by the object of the instance type of the instance expression, in the form `instance_expression.service_name`.
  - The enclosing object, in the form consisting of just a `service name`.
- The instance service declaration is not accessible (see Section 12.9.4, “Compile-Time Processing of Service Invocations”) to the invocation statement.

An *instance service invocation* is executed by first evaluating either:

- The instance expression in the form `instance_expression.service_name`
- The reserved word `this` in the form consisting of just a service name.

The result of this evaluation is known as the target reference. If evaluation of either the instance expression or the reserved word `this` completes abruptly or the evaluation results in a null instance, then the invocation statement completes abruptly for the same reason.

The argument expressions are the evaluated. If the evaluation of any argument expression completes abruptly, then the invocation statement completes abruptly for the same reason. The instance service definition for the target reference is then executed. Finally, if this completes normally. then after it is left, any necessary assigning back of formal to actual parameters occurs.

12.9.4. Compile-Time Processing of Service Invocations

Although the invocation of domain, object and instance service are split in the grammar, the three types of invocation are in many ways inseparable. This is because of the ambiguity between the three that is only resolved by using the current scope of the invocation statement. Determining the service that will be invoked in an invocation statement involves the following steps:

1. If the form is `domain_name::service_name`, then the invocation statement is a domain service invocation.
2. If the form is `object_name::service_name`, then the invocation statement is a object service invocation.
3. If the form is `instance_expression::service_name`, then:

i. If there is an instance service declared by the object, given by the object of the instance type of the instance expression, that is applicable and accessible, the the invocation statement is an instance service invocation.

ii. Otherwise the invocation statement is an object service invocation.

4. If the form consists of just *service name* then:

i. If the invocation statement is in an instance state or an instance service and there is an instance service declared by the enclosing object that is applicable and accessible, then the invocation statement is an instance service invocation.

ii. Otherwise, if the invocation statement is in a state or an object service and there is a non-instance service declared by the enclosing object that is applicable and accessible, then the invocation statement is an object service invocation.

iii. Otherwise, the invocation statement is a domain service invocation.

A service declaration is applicable to a service invocation if and only if both the following are true:

- The number of parameters in the service declaration is the same as the number of argument expressions in the service invocation.
- The type of each actual argument is assignable to the type of the corresponding parameter.

Whether a service declaration is accessible to a service invocation depends upon the access modifier (`public`, `private` or none) in the service declaration in relation to the service invocation.

12.9.5. Argument Lists

The arguments of a service invocation are given in an *argument list*.

[163] argument list = (argument, {(',', argument} ;

[164] argument = expression ;

The number of arguments in an invocation must equal the number of parameters in the services declaration. In addition, the type of each argument must be assignable to the type of the corresponding parameter. If a service is overloaded, the number of

arguments and the compile-time types of the arguments are used, at compile-time, to determine the service that will be invoked (see Section 12.9.4, “Compile-Time Processing of Service Invocations”).

12.10. Blocks

A *block statement* is a sequence of variable declarations followed by a handled sequence of statements.

[165] block statement = ['declare', variable declaration, {variable declaration}], 'begin', handled sequence of statements, 'end', ';' ;

A block statement is executed by executing each of the variable declarations in order from first to last, followed by the execution of the handled sequence of statements. If all of these complete normally, then the block statement completes normally. If any of these complete abruptly for any reason, then the block statement completes abruptly for the same reason.

12.11. The If Statement

The `if` statement allows conditional execution of a sequence of statements or a conditional choice of several sequence of statements, executing one only.

[166] if statement = 'if', condition, 'then', sequence of statements, {('elsif', condition, 'then', sequence of statements)}, [('else', sequence of statements)], 'end', ['if'], ';' ;

An `if` statement is executed by evaluating the first *condition* and then the other *conditions* in succession, until one evaluates to `true`, or all *conditions* evaluate to `false`. If a *condition* evaluates to `true`, then the corresponding sequence of statements is executed; otherwise the sequence of statements following the optional else is executed.

If evaluation of a *condition* completes abruptly for some reason, the `if` statement completes abruptly for the same reason. The `if` statement completes normally only if execution of the selected sequence of statements completes normally.

12.12. The Case Statement

The `case` statement transfers control to one of several sequences of statements depending on the value of an expression.

[167] case statement = 'case', expression, 'is', {case statement alternative}, 'end', ['case'], ';' ;

[168] case statement alternative = 'when', discrete choice list, '=>', sequence of statements ;

[169] discrete choice list = discrete choice, {'|', discrete choice list} ;

[170] discrete choice = constant expression | 'others' ;

All of the following must be true or a compile-time error will result:

- The type of the expression must have an equality operator defined.
- The type of every constant expression associated with a *case statement* must be assignable to the type of the expression.
- No two of the constant expressions associated with a case statement may have the same value.
- At most one `others` discrete choice may be associated with the same case statement and it must appear as the last case statement alternative on its own.

When the *case statement*, is executed, first the expression is evaluated. If evaluation of the expression completes abruptly for some reason, the case statement completes abruptly for the same reason. Otherwise execution continues by comparing the value of the expression with each case statement alternative. Then there is a choice:

- If one of the *constant expressions* is equal to the value of the expression, then we say it matches and the *sequence of statements* after the matching => is executed. If the *sequence of statements* completed normally, then the entire *case statement* completes normally.
- If no *case statement* alternative matches but there is a *case statement* alternative with the reserved word `others`, then the sequence of statements after the matching => is executed. If the *sequence of statements* completed normally, then the entire *case statement* completes normally.
- If no *case statement* alternative matches and there is no case statement alternative with the reserved word `other`, then an exception is raised.

If any statement immediately contained by the *sequence of statements* complete abruptly, the *case statement* completes abruptly for the same reason.

12.13. The While Statement

The `while` statement executes a *sequence of statements* repeatedly until the value of the *condition* is false.

[171] while statement = 'while', condition, 'loop', sequence of statements, 'end', ['loop'], ';' ;

A `while` statement is executed by first evaluating the *condition*. If evaluation of the *condition* completes abruptly for some reason, the `while` statement completes abruptl for the same reason. Otherwise, execution continues by making a choice based on the resulting value:

- If the value is `true`, then the contained *sequence of statements* is executed. Then there is a choice:
  - If the execution of the *sequence of statements* completes normally, then the entire `while` statement is executed again, beginning by re-evaluating the *condition*.
  - If the execution of the *sequence of statements* completes abruptly, see Section 12.13.1, “Abrupt Completion” below.
- If the value of the *condition*. is false,then no further action is taken and the `while` statement completes normally.

If the value of the *condition*. is false the first time it is evaluated, then the *sequence of statements* is not executed.

12.13.1. Abrupt Completion

Abrupt completion of the contained *sequence of statements* is handled in the following manner:

- If execution of the *sequence of statements* completes abruptly because of an `exit` statement, no further action is taken and the `while` statement completes normally.
- If execution of the *sequence of statements* completes abruptly for any other reason, the `while` statement completes abruptly for the same reason.

12.14. The For Statement

The `for` statement executes a *loop parameter specification*, then executes a *sequence of statements* repeatedly until all the values specified by the *loop parameter specification* have been covered.

[172] for statement = 'for', loop parameter specification, 'loop', sequence of statements, 'end', ['loop'], ';' ;

[173] loop parameter specification = variable name, 'in', ['reverse'], range ;

For execution of a `for` statement, the *loop parameter specification* is first elaborated. This elaboration creates the loop parameter and elaborates the *range*. If the *range* defines a empty range, then execution is complete. Otherwise, the *sequence of statements* is executed once for each value of the *range* (or until the `for` statement is left as a consequence of a transfer of control). Prior to each iteration, the corresponding value of the *range* is assigned to the loop parameter. These values are assigned in increasing order unless the reserved word `reverse` is present, in which case the values are assigned in decreasing order.

The *range* shall resolve to some specific integer or enumeration type. If the type of the *range* resolves to a numeric, then the *range* is defined to be of the pre-defined type integer with bounds given by a conversion to integer of bounds of the range.

12.14.1. Abrupt Completion

Abrupt completion of the contained *sequence of statements* is handled in the following manner:

- If execution of the *sequence of statements* completes abruptly because of an `exit` statement, no further action is taken and the `for` statement completes abruptly for the same reason.

12.15. The Exit Statement

An `exit` statement transfers control out of an enclosing `while` statement or `for` statement.

[174] exit statement = 'exit', [('when', condition)], ';' ;

An *exit statement* is executed by evaluating the optional *condition*. If the *condition* evaluates to true or if no condition is present, then the `exit` statement transfers control to the innermost enclosing `while` statement or `for` statement; this statement, which is called the break target, then immediately completes normally.

A compile-time error occurs if there is no `while` statement or `for` statement; that encloses the `exit` statement.

12.16. The Raise Statement

The `raise` statement causes an exception to be raised. The result is an immediate transfer of control.

[175] raise statement = 'raise', [exception specification], ';' ;

The following production is repeated from Section 11.4, “Handling of an Exception” to make the presentation here clearer:

[149] exception specification = domain name, '::', exception name | exception name ;

A compile-time error occurs if:

- The exception to be raised has not been declared by an *exception declaration*, as an exception of either:
  - The domain named by the *domain name*, in the form domain_name`::service_name`.
  - The enclosing domain, in the form consisting of just an exception name.
- A *raise statement* of the form that has no *exception specification*, that is used outside an exception handler.

12.17. The Delete Statement

A `delete` statement allows instances of objects to be deleted.

[176] delete statement = ('delete', name, ';') | ('delete', 'this', ';') ;

When a delete on a *name* is specified the *name* must be an instance or an instance collection type. In each case the delete will set the *name* to a null or empty collection respectively.

When an instance of an object is deleted, it is no longer available to the domain where the object is defined.

If an instance collection is used, then all the instances specified will be deleted. Deletion of an instance will not cause the deletion of any attached relationships. The analyst must explicitly unlink relationships, or the case of associative objects, unassociate relationships before deleting the participating object instances.

A compile-time error occurs if an attempt to delete `this` is made in a non-terminal state. A `delete` statement is executed by evaluating the instance expression or the instance collection expression. If this evaluation completes abruptly, then the `delete` completes abruptly for the same reason; no instances are deleted. The instance or instances determined by the instance expression or instance collection expression are then deleted.

12.18. The Delay Statement

A `delay` statement provides the means to direct the architecture to pause for a period of time defined as duration in seconds.

[177] delay statement = 'delay', expression ';' ;

A compile-time error occurs if the expression does not evaluate to a type of duration.

12.19. The Link Statement

A `link` statement allows a relationship to be created.

[178] link statement = 'link', instance expression, relationship specification, instance expression, [('using', instance expression)], ';' ;

All of the following must be true or a compile-time error will result:

- The *relationship specification* is a valid navigation from the first instance to either an instance or collection of instances of the object that the second instance is an instance of.
- The first and second instances are not associative instances of the relationship being linked.
- If the relationship is an associative relationship then the optional using clause must be used.
- The optional third instance is an associative instance of the *relationship* being linked.

A *link statement* is executed by evaluating all of the instance expressions. If this evaluation completes abruptly, then the *link statement* completes abruptly for the same reason. The *relationship specification* is elaborated and the instances determined are linked. An attempt to link instances that are already linked for the specified *relationship specification* raises an exception. Any *referential attributes* of the instances that are determined by the given relationship are set to the correct values.

12.20. The Unlink Statement

An `unlink` statement allows a relationship to be deleted.

[179] unlink statement = 'unlink', instance expression, relationship specification, instance expression, [('using', instance expression)], ';' ;

All of the following must be true or a compile-time error will result:

- The *relationship specification* is a valid navigation from the first instance to either an instance or collection of instances of the object that the second instance is an instance of.

- The first and second instances are not associative instances of the *relationship* being linked.
- If the relationship is an *associative relationship* then the optional using clause must be used.
- The optional third instance is an associative instance of the *relationship* being linked.

A `unlink` statement is executed by evaluating all of the instance expressions. If this evaluation completes abruptly, then the *unlink statement* completes abruptly for the same reason. The *relationship specification* is elaborated and the instances determined are unlinked. An attempt to unlink instances that are already unlinked for the specified *relationship specification* raises an exception. Any *referential attributes* of the instances that are determined by the given relationship are unset. An attempt to access these values subsequent results in an exception.

12.21. The Associate Statement

The `associate` allows an associative object instance to be added to a relationship between two object instances.

[180] associate statement = 'associate', instance expression, relationship specification, instance expression, 'to', instance expression, ';' ;

All of the following must be true or a compile-time error will result:

- The relationship specification is a valid navigation from the first instance to either an instance or collection of instances of the object that the second instance is an instance of.
- The first and second instances are not associative instances of the relationship being linked.
- The third instance is an associative instance of the relationship being linked.

An `associate` statement is executed by evaluating all the instance expressions. If this evaluation completes abruptly, then the `associate` statement completes abruptly for the same reason. The relationship specification is elaborated and the instances determined are associated.

An attempt to associate an associative instance to two instances that are not already linked raise an exception.

An attempt to associate an instance that is already associated raises an exception.

Any referential attributes of the instances that are determined by the given relationship are set to the correct values.

12.22. The Unassociate Statement

The `unassociate` statement allows an associative object instance to be removed from a relationship between two object instances.

[181] unassociate statement = 'unassociate', instance expression, relationship specification, instance expression, 'from', instance expression, ';' ;

All of the following must be true of a compile-time error will result:

- The relationship specification is a valid navigation from the first instance to either an instance or collection of instances of the object that the second instance is an instance of.
- The first and second instances are not associative instances of the relationship being linked.
- The third instance is an associative instance of the relationship being linked.

An `unassociate` statement is executed by evaluating all of the instance expressions. If this evaluation completes abruptly, then the `unassociate` statement completes abruptly for the same reason. The relationship specification is elaborated and the instance determined are unassociated.

An attempt to unassociate an associative instance that is not associated to the other two instances raise an exception.

Any referential attributes of the instances that are determined by the given relationship are unset. An attempt to access these values subsequently results in an exception.

12.23. The Generate Statement

The `generate` statement generates an event to either an object or a specific instance of an object.

[182] generate statement = 'generate', event specification, '(', argument list, ')', [('to', instance expression)], ';' ;

[183] event specification = (object name, '.', event name) | event name ;

All of the following must be true or a compile-time error will result:

- The signature of the event to be generated, named by the event name together with the argument list must be declared, by an event declaration, as an event of either:
  - The object, in the enclosing domain, named by the object name, in the form `object_name.event_name`.
  - The enclosing object, in the form consisting of just an *event name*.
- If the event is an instance event then the optional instance expression must be provided.
- If the event is either a creation or assigner event event then the optional instance expression must not be used.
- The optional instance expression is an instance of the object that the event belongs to.

A `generate` statement is executed by evaluating the argument expressions. If the evaluation of any argument expression completes abruptly, then the `generate` statement completes abruptly for the same reason. The optional instance expression is then evaluated. If this evaluation completes abruptly, then the `generate` statement completes abruptly for the same reason. The result of this evaluation is known as the target reference. If the optional instance expression is not given, then the target reference is the object of the event. The optional instance expression must be a non-null instance of the object otherwise an exception is raised. The event is then generated to the target reference.

12.24. The Pragma Statement

The `pragma` statement provides the means to define a language pragma within a statement.

[184] pragma = pragma, ';' ;
statement

13. Names

Names are used to refer to declared entities. A declared entity is an attribute, a parameter or a variable. Names can also denote components, elements and attributes of other names, or attributes of the instance denoted by the reserved word `this`.

[185] name = attribute name | parameter name | variable name | indexed component | selected component | selected attribute | slice ;

A compile-time error occurs if:

- An attribute name does not name an attribute of the enclosing object. Note that this will only be possible within an instance state or an instance service.
- A parameter name does not name a parameter of the enclosing state or service.
- A variable name does not name a variable currently in scope.
- `this` is used outside an instance state or an instance service.

Certain forms of *name* (indexed components, selected components and selected attributes) include a prefix that designates some related entity.

The evaluation of a name determines the entity denoted by the name. This evaluation has no other effect for a name that is an attribute name, a parameter name or a variable name.

The evaluation of a name that has a prefix includes evaluation of the prefix. The evaluation of the prefix consists of the evaluation of the name or the reserved word `this`. The type of the prefix is the type of the name or reserved word `this`.

A name may be used as an expression (see Section 14.4.3, “Names as Expressions”).

13.1. Indexed Components

An *indexed component* denotes an element of a sequence, array, string or wstring.

[186] indexed component = prefix, '[', expression, ']' ;

[187] prefix = name ;

A compile-time error occurs if:

- The type of the *prefix* is not a sequence type, an array type, a string type of awstring type.

- The type of the expression is not the corresponding index type.

When the type of the *prefix* is a sequence type, the *indexed component* denotes the element of the sequence with the specified index value. The type of the *indexed component* is the element type of the sequence type.

When the type of the *prefix* is an array type, the *indexed component* denotes the element of the array with the specified index value. The type of the *indexed component* is the element type of the array type.

When the type of the *prefix* is an string type, the *indexed component* denotes the element of the string with the specified index value. The type of the *indexed component* is character. When the type of the *prefix* is a wstring type, the *indexed component* denotes the element of the wstring with the specified index value. The type of the *indexed component* is `wcharacter`. A check is made that the index value belongs to the corresponding index range of the type of the *prefix*. An exception is raised if this check fails.

13.2. Selected Components

A *selected component* is used to denote a component of a structure.

[188] selected component = prefix, '.', component name ;

All of the following must be true or a compile-time error will result:

- The type of the prefix is a structure type.
- The component name names a component of the structure type.

13.3. Selected Attributes

A selected attribute is used to denote an attribute of an instance.

[189] selected attribute = prefix, '.', attribute name ;

All of the following must be true or a compile-time error will result:

- The type of the prefix is an object instance.
- The attribute name names an attribute of the object of the instance type.

A check is made that the value of the prefix is not a null instance. An exception is raised if this check fails.

13.4. Slices

A *slice* denotes a value that is formed by a sequence of consecutive elements of a sequence, array, string or wstring.

[190] slice = prefix, '[', range, ']' ;

A compile-time error occurs if:

- The type of the prefix is not a sequence type, an array type, a string type or a wstring type.
- The set of values defined by the range is not a subset of the set of values of the corresponding index type.

For a prefix whose type is a sequence, string or wstring type, the type of a *slice* is the type of the prefix. For a prefix whose type is an array type, the type of a *slice* is the sequence type whose element type is the same as the element type of the prefix.

A *slice* denotes a value that is formed by the sequence of consecutive elements of a sequence, array, string or wstring denoted by the prefix, corresponding to the range of values of the index given by the range.

If the *slice* is not a null slice (a *slice* where the range is a null range), then a check is made that the set of values given by the range belong to the index range of the sequence, array, string or wstring denoted by the prefix. An exception is raised if this check fails.

13.5. Assignability of Names

When a name is used on the left-hand side of an assignment or on the right-hand side of an input stream, a check is made that the name is assignable. If it is not a compile-time error occurs. An attribute name is assignable if the attribute it names is not preferred or referential. A parameter name is assignable if the parameter it names is of mode out. A variable name is assignable if the variable it names is not read only. `this` is not assignable. An indexed component is assignable if its prefix is assignable. A selected component is assignable if its prefix is assignable. A selected attribute is assignable if the attribute it names is not preferred or referential.

14. Expressions

The rules applicable to the different forms of expression and to their evaluation, are given in this chapter.

14.1. Evaluation and Result

When an expression is evaluated, the result denotes a value. Each expression occurs in the declaration of some type that is being declared or in a body of a state or service.

14.2. Type of an Expression

An expression has a type known at compile-time. The rules for determining the type of an expression are explained separately below for each kind of expression.

14.3. Normal and Abrupt Completion of Evaluation

Every expression has a mode of evaluation in which certain computational steps are carried out. The following sections describe the normal mode of evaluation for each kind of expression. If all the steps are carried out without an exception being raised, the expression is said to complete normally. If, however, evaluation of an expression raises an exception, then the expression is said to complete abruptly. An abrupt completion always has an associated reason, which is always an exception with a given value.

Run-time exceptions are thrown by the pre-defined operators as follows:

- A create expression or concatenation operator expression raises an exception if there is insufficient memory available.
- A selected attribute expression raises an exception if the value of the instance prefix is `null`.
- An indexed component expression raises an exception if the value of the index expression is not within the bounds of the corresponding index type.
- A type conversion raises an exception if there is no value of the target type that corresponds to the operand value.
- An integer division or integer remainder raises an exception if the value of the right hand operand expression is zero.

I an exception occurs, then evaluation of one or more expressions may be terminated before all steps of their normal mode of evaluation are complete; such

expressions are said to complete abruptly. The terms "complete normally" and "complete abruptly" are also applied to the execution of statements. A statement may complete abruptly for a variety of reasons, not just because an exception is raised.

If evaluation of an expression requires evaluation of a sub-expression, abrupt completion of the sub-expression always causes the immediate abrupt completion of the expression itself, with the same reason and all succeeding steps in the normal mode of evaluation are not performed.

14.4. Primary Expressions

Primary expressions include most of the simplest kinds of expressions, from which all others are constructed. A parenthesized expression is also treated syntactically as a *primary expression*.

[191]	primary expression	=	literal | 'this' | name as expression | characteristic reference | aggregate | service as expression | function invocation | event as expression | '(', expression, ')' ;
[192]	aggregate	=	structure aggregate | object aggregate ;

14.4.1. Literals

A literal represents a value literally, that is, by means of a notation suited to its kind. The following production is repeated from Section 2.6, “Literals” to make the presentation clearer.

[14]	literal	=	numeric literal | character literal | string literal | boolean literal | enumerator literal | null literal | device literal | stream literal ;

The type of a literal is determined as follows:

- The type of a numeric literal is numeric.
- The type of a character literal is wcharacter.
- The type of a string literal is string.
- The type of a boolean literal is boolean.
- The type of a enumeration literal is its *enumeration type*.
- The type of the null literal is instance.

- The type of the device literal is device.

Evaluation of a *literal* yields the corresponding value of the type. Evaluation of a *literal* always completes normally.

14.4.1.1. Enumeration Literals

An *enumeration literal* represents a value of an enumeration type literally, that is, by means of the name given to one of its kind. The following production is repeated from Section 3.7.3, “Enumeration Types” to make the presentation clearer.

[68] enumerator literal = enumerator name | type name, '.', enumerator name | domain name, '::', enumerator name | domain name, '::', type name, '.', enumerator name ;

A compile-time error occurs if any of the following are true:

- In the form consisting of just an enumerator name, the enumerator name does not name an enumerator of an enumeration type in the enclosing domain.
- In the form *type_name*.*enumerator_name*, the enumerator name does not name an enumerator of an enumerator type in the enclosing domain.
- In the form *domain_name*.*enumerator_name*, the enumerator name does not name an enumerator of an enumerator type in the domain named by the domain name.
- The type of the enumerator is not accessible to the current scope.

The the same enumerator name is specified for more than one enumerator type definition in the same domain, the corresponding *enumeration literals* are said to be overloaded. If either of the the two forms of *enumeration literal* that don't specify a type name are used, this can cause an ambiguity. This ambiguity will be resolved if the *enumeration literal* is used as either:

- Part of a binary operation where the type of the other expression is not ambiguous.
- As part of an assignment.

If the ambiguity cannot be resolved a compile-time error occurs.

Evaluation of an *enumeration literal* yields the corresponding value of the type. Certain forms of *enumeration literal* include a type name or a domain name of both. These forms are used to:

- Access enumerators from other domains, as in `Calendar::MONDAY` and Calendar::day_name_type.
- Remove any ambiguity when two or more, enumerator types have enumerators of the same name, as in `day_name_type.MONDAY` and `Calendar::day_name_type.MONDAY`

Use of these qualified forms, where the domain given is the enclosing domain or where no ambiguity exists, is allowed.

14.4.2. this

The reserved word `this` may be used in either the body of an object instance service (see Section 9.2.1.3, “Instance Object Functions” and Section 10.2.1.3, “Instance Object Services”) or the body of an instance state (see Section 8.1, “States”. If it appears anywhere else, a compile-time error occurs.

The type of `this` is the instance> type of the object within which the reserved word `this` occurs. When used as a primary expression, the reserved word `this` denotes a value that is a reference to the instance either for which the object instance service was invoked or which has just moved into the current state.

14.4.3. Names as Expressions

The value of a primary expression that is a name, is the value of the entity that the name denotes.

[193] name as expression = name, ;

14.4.4. Characteristic References

A *characteristic reference* is a characteristic of an entity that can be queried.

[194] characteristic reference = name, "'", characteristic reference, [('(', argument list, ')'] | type, "'", characteristic reference, [('(', argument list ;

[195] characteristic name = identifier ;

A compile-time error occurs if any of the following is true:

- In the form *name*'*characteristic_name*, the type of the name does not define a value characteristic named by the *characteristic name*.
- In the form *type*'*characteristic_name*, the type does not define a type characteristic named by the *characteristic name*.

The type of a characteristic reference is defined by either the type of the name or the type. Chapter 3, *Types* defines the value and type characteristics for all types.

14.4.5. Structure Aggregates

A *structure aggregate* combines component values into a composite type value of a structure type.

[196] structure aggregate = '(', expression, ',', expression, {(',', expression)}, ')' ;

For the evaluation of a *structure aggregate*, an anonymous structure is created and values for the components are obtained and assigned into the corresponding components of the anonymous structure. The value of the *structure aggregate* is the value of this anonymous structure. Applying an operator to the value or assigning the value to an entity, checks that the anonymous structure can be converted into a structure of an appropriate type. If this is not possible, a compile-time error occurs.

14.4.6. Services As Expressions

A service expression is used to represent an object service.

[197] service as expression = domain name, '::', object name, '.', service name | domain name, '::', service name | object name, '.', service name | service name ;

A compile-time error occurs if the service is not declared, by an service declaration, as a service of either:

- The domain named by the domain name, in the form *domain_name*::*service_name* or the enclosing domain, in the form consisting of just a *service name*.
- The object named by the object name, in the domain named by the domain name, in the form *domain_name*::*object_name*.*service_name*.
- The object named by the object name, in the enclosing domain, in the form *object_name*.*service_name*.
- The enclosing object, in the form consisting of just a *service name*.

14.4.7. Events As Expressions

An *event as expression* represents an event.

[198] event as expression = object name, '.', event name | event name, ;

A compile-time error occurs if the event is not declared by an event declaration, as an event of either:

- the object, in the enclosing domain, named by the object name, in the form object_name.event_name or
- the enclosing object, in the form consisting of just an event name.

14.4.8. Function Invocation

A function invocation as an expression is the evaluation of the invocation of the function. The type of the expression is given by the type of the return parameter.

[199] function invocation = domain function invocation | object function invocation | instance function invocation ;

14.4.9. Domain Function Invocation

A *domain function invocation* is used to invoked a domain function.

[200] domain function invocation = domain name, '::', function name, '(', argument list, ')',';' | function name, '(', argument list, ')',';' ;

A compile-time errors occurs if either of the following is true:

- The signature of the *domain function invocation* to be invoked, named by the function name together with the argument list has not been declared, by a domain function declaration, as a function of either:
  - The domain named by the domain name, in the form *domain_name*::*function_name*.
  - The enclosing domain, in the form consisting of just a *function name*.
- The domain function declaration is not accessible (see Section 14.4.12, “Compile-Time Processing of Function Calls”) to the invocation statement.

A *domain function invocation* is executed by evaluating the argument expressions. If the evaluation of any argument expression completes abruptly, then the *domain function invocation* completes abruptly for the same reason. Otherwise, the *domain function invocation* is then executed. If this complete normally, the evaluation of the expression return parameter is performed.

14.4.10. Object Function Invocation

A *object function invocation* is used to invoked a domain function.

[201] object function invocation = object name, '.', function name, '(', argument list, ')',';' | instance expression, '.', function name, '(', argument list, ')',';' | function name, '(', argument list, ')',';' ;

A compile-time errors occurs if either of the following is true:

- The signature of the *object function invocation* to be invoked, named by the function name together with the argument list has not been declared, by an object function declaration, as a non-instance function of either:
  - The object, in the enclosing domain, named by the object name, in the form *object_name*.*function_name*.
  - The object, given by the object of the instance type of the instance expression, in the form *instance_expression*.*function_name*.
  - The enclosing object, in the form consisting of just a *function name*.
- The object function declaration is not accessible (see Section 14.4.12, “Compile-Time Processing of Function Calls”) to the invocation statement.

A *object function invocation* is executed by evaluating the argument expressions. If the evaluation of any argument expression completes abruptly, then the *object function invocation* completes abruptly for the same reason. Note, that for an *object function invocation* of the form *instance_expression*.*function_name*, the instance expression is not evaluated. If the argument expression complete normally, the *object function definition* is then executed. If this complete normally, the evaluation of the expression return parameter is performed.

14.4.11. Instance Function Invocation

An *instance function invocation* is used to invoke an object instance function.

[202] instance function invocation = instance expression, '.', function name, '(', argument list, ')',';' | function name, '(', argument list, ')',';' ;

A compile-time errors occurs if either of the following is true:

- The signature of the *object instance function invocation* to be invoked, named by the function name together with the argument list has not been declared, by an object function declaration, as a instance function of either:

- The object, given by the object of the instance type of the instance expression, in the form *instance_expression*`.`*function_name*.
- The enclosing object, in the form consisting of just a *function name*.

- The object instance function declaration is not accessible (see Section 14.4.12, “Compile-Time Processing of Function Calls”) to the invocation statement.

A *object instance function invocation* is executed by evaluating either:

- The instance expression in the form *instance_expression*`.`*function_name*.
- The reserved word `this` in the form of just a *function name*.

The result of this evaluation is known as the target reference. If evaluation of either the instance expression or the reserved word `this` completes abruptly or the evaluation results in the `null` instance, then the function call completes abruptly for the same reason. The argument expressions are then evaluated. If the evaluation of any argument expression completes abruptly, then the function call completes abruptly for the same reason. The *object instance function definition* for the target reference is then executed. If this completes normally. The evaluation of the expression return parameter is then performed.

14.4.12. Compile-Time Processing of Function Calls

Although the invocation of domain, object and instance functions are split in the grammar, the three type of invocation are in may ways inseparable. This is because of the ambiguity between the three types, that is only resolved using the current scope of the invocation statement. Determining the function that will be invoked in a function call involves the following steps:

1. If the form is *domain_name*`::`*function_name*, then the function call is a domain function invocation.
2. If the form is *object_name*`.`*function_name*, then the function call is a domain function invocation.
3. If the form is *instance_expression*`.`*function_name*, then:
   - If there is an instance function declared by the object, given by the object of the instance type of the instance expression, that is applicable and accessible, then the function call is a *instance function invocation*.

- Otherwise the function call is an *object function invocation*.

4. If the form consists of just a *function name*, then:

   - If the function call is in an instance state or instance service and there is an instance function declared by the enclosing object that is applicable and accessible, then the function call is an instance function invocation.

   - If the function call is in a state or an object service and there is a non-instance function declared by the enclosing object that is applicable and accessible, the the function call is an object function invocation.

   - The invocation statement is a domain function invocation.

A function declaration is applicable to a function call if and only if both of the following are true:

- The number of parameters in the function declaration is the same as the number of argument expressions in the function invocation.

- The type of each actual argument is assignable to the type of the corresponding parameter.

Whether a function declaration is accessible to a function invocation depends upon the access modifier (see Section 9.1.1, “Domain Function Modifiers” and Section 9.2.1, “Object Function Modifiers”) in the function declaration in relation to the function call.

14.4.13. Parenthesized Expressions

A parenthesized expression is a primary expression whose type is the type of the contained expression and whose value is the value of the contained expression.

14.5. Unary Operators

The unary operators are `+`, `-`, `not` and `abs`. Expressions with unary operators group right-to-left, so that `-+`*x* has the same meaning as `-(+`*x*`)`.

[203] unary expression = type conversion | '-', unary expression | '+', unary expression | 'not', unary expression | 'abs', unary expression | primary expression ;

Evaluation of an unary operator expression first evaluates the unary expression, then its operand and finally applies the operator.

14.5.1. Unary Plus and Minus Operators

A compile-time error occurs if the type of the expression of the + and - operators is not a numeric type. The type of an unary + or - expression is the type of the operand. Unary + and - have their conventional meaning.

14.5.2. Logical Negation Operator

A compile-time error occurs if the type of the expression of the `not` operator is not a boolean type. The unary `not` operator yields the logical negation of the operand. Hence, its value is true if its operand is false, and false if its operand is true.

14.5.3. Absolute Value Operator

A compile-time error occurs if the type of the expression of the `abs` operator is not a numeric type. The type of the unary `abs` expression is the type of the operand. The unary `abs` operator yields the absolute value of the operand.

14.6. Type Conversion

A type conversion converts a value from one type to a value of another type. Whether one type is convertible to a second type is defined in Chapter 4, *Type Conversion*.

[204] type conversion = type, '(', expression, ')' ;

A compile-time error occurs if the type of the unary expression is not convertible to the specified type. The type of the type conversion is the type whose name appears within the parentheses. The parentheses and the type they contain are sometimes called the coercion operator. The result of a type conversion is a value of the unary expression in the specified type. At run-time, the operand value is converted to the type specified by the coercion operator.

For evaluation of a type conversion, the operand is evaluated and then the value of the operand is converted to a corresponding value of the target type. If there is no value of the target type that corresponds to operand value, an exception is raised. If the result of the conversion fails to satisfy a constraint imposed by the target type, then an exception is raised.

14.7. Multiplicative Operators

The operators `*`, `/`, `mod`, `**`, `rem`, `intersection`, `intersection` and `disunion` are called the multiplicative operators. They have the same precedence and are syntactically left-associative and group left-to-right.

[205] multiplicative expression = unary expression | multiplicative expression, '*', unary expression | multiplicative expression, '/', unary expression | multiplicative expression, 'mod', unary expression | multiplicative expression, '**', unary expression | multiplicative expression, 'rem', unary expression | multiplicative expression, 'intersection', unary expression | multiplicative expression, 'disunion', unary expression ;

A compile-time error occurs if any of the following are true:

- The type of both operands of the `*`, `/`, `mod`, `**` and `rem` operators are not types of subtype of a numeric type.
- The types of both operands of the `intersection` and `disunion` in the operators are not type or subtypes of a set type.
- For the `*` and `/` operators, the type of the right-hand operand is not convertible to the type of the left-hand operand.
- For all the other operators, the type of the right-hand operand is not assignable to the type of the left-hand operand.

14.7.1. Multiplication and Division Operators

If the type of the right-hand operand is assignable to the type of left-hand operand then the type of a `*` or `/` expression is the base type of the left-hand side operand. Otherwise, it is the basis type of the left-hand side operand.

The `*` operator performs multiplication, producing the product of the operands.

The `/` operator performs division producing the quotient of the operands.

The multiplication and division operators for floating-point types have their conventional meaning. The accuracy of the result is determined by the precision of the result type. Signed integer division has it's conventional meaning. Signed integer division and remainder are defined by the relation:

```
A= (A/B) * B + (A rem B)
```

where (*A* `rem` *B*) has the sign of *A* and an absolute value less than the absolute value of *B*.

Signed integer division the identity:

```
(-A/B) = -(A/B) = A/(-B)
```

An exception is raised if the right-hand side operand of a division expression is zero.

14.7.2. Remainder and Modulus Operators

The type of a `rem` or `mod` expression is the base type of the left-hand side operand. The `rem` yields the remainder of the operand from an implied division. The signed integer modulus operator is defined such that the result of *A* `mod` *B* has a sign of *B* and an absolute value less then the absolute value of *B*. In addition, for some signed integer value *N*, this result satisfies the relation:

```
A = B*N + (A mod B)
```

An exception is raised if the right-hand side operand of either a `rem` or a `mod` expression is zero.

The Table 14.1, “Relations between `/`, `rem` and `mod`” gives examples of the use of the `rem` and `mod` operators.

Table 14.1. Relations between `/`, `rem` and `mod`

A	***B***	***A*/*B***	***A*rem*B***	***A*mod*B***
10	5	2	0	0
11	5	2	1	1
12	5	2	2	2
13	5	2	3	3
14	5	2	4	4
10	-5	-2	0	0
11	-5	-2	1	-4
12	-5	-2	2	-3
13	-5	-2	3	-2
14	-5	-2	4	-1
-10	5	-2	0	0
-11	5	-2	-1	4
-12	5	-2	-2	3
-13	5	-2	-3	2
-14	5	-2	-4	1
-10	-5	2	0	0
-11	-5	2	-1	-1

A	*B*	*A/B*	*A*rem*B*	*A*mod*B*
-12	-5	2	-2	-2
-13	-5	2	-3	-3
-14	-5	2	-4	-4

14.7.3. Exponentiation Operator

The type of a ** expression is the base type of the left-hand side operand. The ** operator performs exponentiation, producing the value of the left-hand operand multiplied by itself *n*, where *n* is the value of the right-hand operand.

14.7.4. Set Intersection and Disunion Operators

The type of a intersection and disunion expression is the base type of the left-hand side operand. The base type of a set type is the set type whose element type is the base type of the original element type. The `intersection` and `disunion` operators yield the set intersection and set disunion of the operands.

14.8. Additive Operators

The operators +, -, &, `union` and `not_in` are called the additive operators. They have the same precedence and are syntactically left-associative and group left-to-right.

[206] additive expression = multiplicative expression | additive expression, '+',
multiplicative expression | additive expression, '-',
multiplicative expression | additive expression, '&',
multiplicative expression | additive expression, 'union',
multiplicative expression | additive expression, 'not_in',
multiplicative expression ;

A compile-time error occurs if any of the following are true:

- The types of both operands of the + and - operators are not types or subtypes of a numeric type.
- The types of both operands of the & operator are not types or subtypes of string, wstring, set, bag or sequence types.
- The types of both operands of the `union` and `not_in` operators are not types or subtypes of a set type.
- The type of the right-hand operand is not assignable to the type of the left-hand side operand.

14.8.1. Additive Operators for Numeric Types

The type of a + or - expression for numeric types is the base type of the left-hand side operand. The + and - operators for numeric types have their conventional meaning.

14.8.2. String Concatenation Operator

The type of a *string concatenation* expression is the base type of the left-hand side operand. String concatenation yields a new string that is the concatenation of the operand strings.

14.8.3. Collection Addition Operator

The type of a collection addition expression is the base type of the left-hand side operand. The base type of a set type is the set type whose element type is the base type of the original element type. The base type of a bag type is the bag type whose element type is the base type of the original element type. The base type of a sequence type is the sequence type whose element type is the base type of the original element type. Collection addition yields a new collection that contains all the elements of both operands:

- For sets, duplicates are removed and there is no order.
- For bags, duplicates are preserved and there is no order.
- For sequence, duplicates are preserved and the order is also preserved.

14.8.4. Set Union and Not In Operators

The `union` and `not_in` expression is the base type of the left-hand side operand. The base type of a set type is the set type whose element type is the base type of the original element type. The `union` and `not_in` operators yield the set union and set subtraction of the operands.

14.9. Relational Operators

The operators < , > ,<= and >= are called relational operators.

[207] relational expression = additive expression | additive expression, '<', additive expression | additive expression, '>', additive expression | additive expression, '<=', additive expression | additive expression, '>=', additive expression ;

A compile-time error occurs if any of the following are true:

- The types of both operands are not types or subtypes of character, wcharacter, string, wstring, numeric or enumeration types.
- The type of the right-hand operand is not assignable to type of the left-hand operand.

The type of a relational expression is always boolean. The relational operators for meaning is as follows:

- The relational operators for two character or wcharacter values have their conventional numeric meaning.
- The relational operators for two string or wstring values have their conventional lexicographical meaning.
- The relational operators for two numeric values have their conventional lexicographical meaning.
- The relational operators for two enumeration have their meaning defined by the order in which the enumerators of an enumeration type are defined (see Section 3.7.3, “Enumeration Types”).

14.10. Equality Operators

The = (equal to) and the /= (not equal to) operators are analogous to the relational operators except for their lower precedence. Thus, *a*<*b*=*c*<*d* is true whenever *a*<*b* and *c*<*d* have the same truth value.

```
[208]          equality  =  relational expression | relational expression, '/=',
             expression      relational expression | relational expression, '=',
                             relational expression ;
```

The equality operators may be used to compare two operands of the compatible types. A compile-time error occurs if any of the following are true:

- The types of both operands are not types or subtypes of built-in, instance, numeric or enumeration types.
- The type of the right-hand operand is not assignable to the type of the left-hand operand.

The *equality expression* are not defined for structure or collection types. The type of an *equality expression* is always boolean.

Two character, wcharacter, boolean, byte, numeric or enumeration values are equal if they have the same value.

Two string or wstring values are equal if they contain the same sequence of elements in the same order.

Two instances are equal if they refer to the same instance.

In all cases, *a*`/=`*b* produces the same result as `not(`*a*`=`*b*`)`.

14.11. Logical Operators

The logical operators are the and operator `and`, the exclusive-or operator `xor` and the inclusive-or operator `or`. These operators have different precedence, with `and` having the highest precedence and `or` the lowest precedence. Each of these operators is syntactically left-associative and each groups left-to-right. Each operator is commutative and associative.

[209]	logical and expression	=	equality expression | logical and expression, 'and', equality expression ;
[210]	logical xor expression	=	logical and expression | logical xor expression, 'xor', logical and expression ;
[211]	logical or expression	=	logical xor expression | logical or expression, 'or', logical xor expression ;

A compile-time error occurs if and of the following is true:

- The types of both operands are not types or subtypes of boolean.
- The type of the right-hand operand is not assignable to the type of the left-hand operand.

The type of a logical expression is the base type of the left-hand operand. For `and`, the result is true if both operand values are true; otherwise, the result is false. For `xor`, the result is true if the operand values are different; otherwise, the result is false. For `or`, the result is false if both operand values are false; otherwise, the result is true.

14.11.1. Short Circuit Evaluation of Logical Operators

The `and` operator evaluates is right-hand operand only if the value of its left-hand operand is true. It is syntactically left-associative and groups left-to-right. It is fully associative with respect to both side effects and result value; that is, for any expressions *a*, *b* and *c*, evaluation of the expression `((`*a*`) and (`*b*`)) and (`*c*`)` produces the same result as evaluation of the expression `(`*a*`) and ((`*b*`) and`

(*c*)). At run-time, the left-hand operand is evaluation first; if its value is false, the value of the expression is false and the right-hand operand expression is not evaluated. If the value of the left-hand operand is true, then the right-hand expression is evaluated and its value becomes the value of the expression.

The `or` operator evaluates its right-hand operand only if the value of its left-hand operand is false. It is syntactically left-associative (it groups left-to-right). It is fully associative with respect to both side effects and result value; that is, for any expressions a, b and c, evaluation of the expression ((*a*) `or` (*b*)) `or` (*c*) produces the same result as evaluation of the expression (*a*) `or` ((*b*) `or` (*c*)). At run-time, the left-hand operand expression is evaluated first; if its value is true, the value of the expression is true and the right-hand operand expression is not evaluated. If the value of the left-hand operand is false, then the right-hand expression is evaluated and it value becomes the value of the expression.

14.12. Expression

An expression is any logical or expression.

[212] expression = logical or expression ;

14.13. Conditions

Conditions are syntactically any *expression*.

[213] condition = expression ;

A compile-time error occurs is the type of the expression is not assignable to boolean. The type of a condition is the type of the expression. A condition yields the value of the expression.

14.14. Instance and Instance Collection Expressions

Instance expressions are syntactically any name of the reserved word `this`. Instance collections are syntactically any name.

[214] instance = name | 'this' ;
expression

[215] instance = name ;
collection
expression

A compile-time error occurs if in an instance expression, the type of the name is not an instance type. A compile-time error also occurs if in an *instance collection expression*, the type of the name is not a collection type whose component type is an instance type. The type of an *instance expression* and an *instance collection*

expression is the type of the name or the reserved word `this`. An *instance expression* and an *instance collection expression*, both yield the value of the name or the reserved word `this`.

14.15. Constant Expressions

Certain expression are defined to be constant. Constant means determinable at compile-time, using the declared properties or values of entities. *Constant expressions* are syntactically any expression.

[216] constant = expression ;
expression

The type of a *constant expression* is the type of the expression. A *constant expression* yields the value of the expression.

A *constant expression* is an expression that is composed using only the following:

- A numeric, character, string, boolean, enumeration or null literal.
- A name that denotes a read only variable whose initializer is a *constant expression*.
- An indexed component whose prefix and expression are both *constant expressions*.
- A selected component whose prefix is a *constant expression*.
- A slice whose prefix and range are both constant.
- A structure aggregate whose component values are all *constant expressions*.
- A *constant expressions* enclosed in parentheses.
- A type conversion whose operand is a constant expression.
- An unary `+`, `-`, `not` or `abs` operator whose operand is a *constant expression*.
- A multiplicative `*`, `/`, `mod`, `**`, `rem`, `intersection` or `disunion` operator whose operands are both *constant expressions*.
- An additive `+`, `-`, `&`, `union` or `not_in` operator whose operands are both *constant expressions*.
- A relational `<`, `>`, `<=` or `>=` operator whose operands are both *constant expressions*.

- An equality = or /=, operator whose operands are both *constant expressions*.
- A logical `and`, `xor` or `or` operator whose operands are both *constant expressions*.

14.16. Executable UML Expressions

All of the types of expression described so far have been type of expression that are not specific to executable UML models. An extended expression is any of the types of expression described so far, together with the types of expression that are specific to executable UML models.

[217] extended expression = create expression | find expression | relationship navigation | correlated relationship navigation | ordering expression | expression ;

The places where extended expression can be used is restricted to the right-hand side of an assignment. In addition, the results obtained from extended expression must be assigned to a variable or parameter and hence must be given a name.

14.16.1. Create Expressions

A *create expression* is used to create new instances of objects.

[218] create expression = 'create', '.', ['unique'], object name, '(', object aggregate, ')' ;

For create expressions where the optional `unique` is not used, the object aggregate must supply initial values for all non-referential preferred attributes, otherwise a compile-time error occurs.

For create expressions where the optional `unique` is used, any non-referential attributes that are not given initial values, are given by the architecture, suitable values that make the created instance unique. If suitable values cannot by found, then an exception is raised.

A compile-time error occurs if the object name does not name an object that has already been declared in the enclosing domain.

The type of a *create expression* is the instance type of the object designated by the object name. For the evaluation of a *create expression*, the elaboration of the object name is performed first. An instance of the designated object is created and values for the attributes are obtained and assigned into the corresponding attributes of the instance. The create expression yields a value that designates the created instance. The only exception to this is in the body of a creation state of the object. If an initial value for the state of the created instance is not given, then the initial state of the created instance is set to the enclosing creation state.

14.16.1.1. Object Aggregates

An *object aggregate* gives initial values for attributes and the state of the created instance in a create expression.

[219] object aggregate = attribute association, {(',', attribute association)} ;

[220] attribute association = attribute name, '=>', expression | 'Current_State', '=>', state name ;

All of the following must be true or a compile-time error will result:

- The attribute name, in an attribute association, names an attribute of the object being created.
- The type of the expression, in an attribute association, is assignable to the type of the attribute named by the attribute name.
- The attribute named by the attribute name, in an attribute association, is assignable (see Section 13.5, “Assignability of Names”).
- The state name, in an attribute association, names an instance or creation state of the object being created.
- There is only one attribute association for an attribute.
- There is only one attribute association for the `Current_State`.
- If an attribute is of a type which is an unconstrained array type, then an attribute association must exit to formally constrain the attribute.

14.16.2. Find Expressions

Instances of an object can be obtained using a *find expression*.

[221] find expression = 'find', find specification, '(', find condition, ')' | 'find_all', find specification, '(', ')' | 'find_one', find specification, '(', [find condition], ')' | 'find_only', find specification, '(', [find condition], ')' ;

[222] find specification = scoped object name | instance collection expression ;

[223] scoped object name = scoped name ;

There are two kinds of *find expression* The first, finds all the instances of an object that satisfy a condition. The second, finds all the instances in an instance collection that satisfy a condition. For both kinds of *find expression* there are four forms:

- `find` finds a set of instances satisfying a condition.
- `find_all` finds all the instances.
- `find_one` finds an arbitrary instance satisfying an optional condition.
- `find_only` finds the only instance satisfying an optional condition.

A compile-time error occurs if the *scoped name* does not name an object that has already been declared.

The evaluation of a find expression proceeds in two steps:

1. First a target set is constructed. For a *find expression* whose find specification is either:
   - *Scoped object name*, the target set contains all the instances of the designated object.
   - An instance instance collection expression, the target set contains all the instances of the collection.
2. The instances of this target set are then used to yield the result of the *find expression*. For results for each type of *find expression* are as follows:
   - For `find`, the *find expression* yields all the instances in the target set that satisfy the find condition.
   - For `find_all`, the *find expression* yields all the instances in the target set.
   - For `find_one`, the *find expression* yields an arbitrary instance in the target set. that satisfies the optional find condition. If there are no instances the satisfy the optional condition, then the *find expression* yields a `null` instance.
   - For `find_only`, the *find expression* yields the only instance in the target set that satisfies the optional find condition. If there are more an one instance the satisfy the optional condition, then an exception is raised.

The type of a `find` and `find_all` expression is the set type whose element type is the instance type of the object being found.

14.16.2.1. Find Conditions

A *find condition* is used to specify which instances are found by a find expression.

[224]	find condition	=	find xor condition | find condition, 'or', find xor condition ;
[225]	find xor condition	=	find and condition | find xor condition, 'xor', find and condition ;
[226]	find and condition	=	find equality condition | find and condition, 'and', find equality condition ;
[227]	find equality condition	=	find relational condition | name, '/=', relational expression | name, '=', relational expression ;
[228]	find relational condition	=	find unary condition | name, '<', additive expression | name, '>', additive expression | name, '<=', additive expression | name, '>=', additive expression ;
[229]	find unary condition	=	'not', find unary condition | '(', find condition, ')' ;

A *find condition* is a restricted condition that is evaluated in the scope of the instance being checked. This means that in addition to all the entities that were in scope, all of the attributes of the instance being checked are also in scope.

A compile-tome error occurs if the name on the left-hand side does not name an attribute or sub-component (using either indexed or selected component) of an attribute of the object being found. The type of a *find condition*, *find xor condition*, *find and condition*, *find equality condition*, *find relational condition* and *find unary condition* is always boolean. For evaluation of a *find condition* for a specific instance, the *find condition* is evaluated on the instance. The *find condition* yields either true or false depending upon whether the instance satisfies the *find condition*.

14.16.3. Relationship Navigation

A *relationship navigation* allows relationships specified between objects to be read in order to determine the instance or collection of instances that are related to an instance or collection of instances of interest.

[230]	relationship navigation	=	instance expression, '->', relationship specification, {('->', relationship specification)} | instance collection expression, '->', relationship specification, {('->', relationship specification)} ;

A compile-time error occurs if the relationship specification is invalid for the type being navigated from. The rules for this are described in Section 7.4, “Relationship Specification”. The type of relationship navigation is described in Section 7.4, “Relationship Specification”. A *relationship navigation* yields either the instance or collection of instances that the given instance or collection of instances are related

to using the relationship specification. equivalent to doing the first navigation, taking the result of this and then doing the second navigation and so on. If a navigation is performed from a `null` instance then result is either a `null` instance or collection of instances.

14.16.4. Correlated Relationship Navigation

A *correlated navigation* navigation allows associative relationships specified between objects to be read in order to determine the instance or set of instances of the associative object that are related to a pair of related instances.

[231] correlated relationship navigation = instance expression, 'with', instance expression, '->', relationship specification ;

A compile-time error occurs if the correlated relationship specification is invalid for the two instance types being navigation from. The rules for this are described in Section 7.5, “Correlated Relationship Specification”. Evaluation of a correlated relationship specification consists of evaluation of both of the instance expressions. The correlated relationship navigation yields either the instance or set of instances that the given instances are related to, using the correlated relationship specification.

14.16.5. Ordering Expressions

An ordering expression allows the analyst to order a collection of instances or a collection of structures.

[232] ordering expression = instance ordering expression | structure ordering expression ;

14.16.5.1. Instance Ordering Expressions

The elements in an instance collection can be explicitly ordered using an *instance ordering expression*.

[233] instance ordering expression = instance collection expression, 'ordered_by', '(', instance ordering, ')' | instance collection expression, 'reverse ordered_by', '(', instance ordering, ')' | find expression, 'ordered_by', '(', instance ordering, ')' | find expression, 'reverse_ordered_by', '(', instance ordering, ')' ;

[234] instance ordering = attribute name, {(',', attribute name)} ;

A compile-time error occurs if:

- The type of the find expression is not an instance collection type.
- An attribute name in an *instance ordering* does not name an attribute of the object whose instances are being ordered.
- An attribute is named more than once in an *instance ordering*.

The type of an *instance ordering* expression is the sequence type whose element type is the element type of the collection being ordered. Evaluation of an *instance ordering* expression first evaluates the instance collection expression or find expression. Evaluation of an *instance ordering* yields a sequence that contains all the elements of the resultant collection in a specific order. In the case of `ordered_by`, the ordering will be such that for each successive element in the sequence, the first will be the element with lowest value of the first attribute in the instance ordering. In the case of `reverse_ordered_by`, the ordering will be such that for each successive element in the sequence, the first will be the element with highest value of the first attribute in the instance ordering. Where ordering on multiple attributes is specified, the sequence will be sorted by the first attribute and then within each value of this, by the second attribute and so on.

14.16.5.2. Structure Ordering Expressions

The components in a structure collection can be explicitly ordered using a *structure ordering expression*.

[235] structure ordering expression = prefix, 'ordered_by','(', structure ordering, ')' | prefix, 'reverse_ordered_by','(', structure ordering, ')' ;

[236] structure ordering = component name, {(',', component name)} ;

A compile-time error occurs if:

- The type of the prefix is not a structure collection type.
- A component name in a *structure ordering* does not name a component of the structure whose values are being ordered.
- A component is named more than once in a structure ordering.

The type of a *structure ordering expression* is the sequence type whose element is the element type of the collection being ordered. Evaluation of a *structure ordering expression* first evaluates the prefix. Evaluation of a *structure ordering expression*

yields a sequence that contains all the elements of the resultant collection in a specific order.

In the case of `ordered_by`, the ordering will be such that for each successive element in the sequence, the first will be the element with the lowest value of the first component in the structure ordering. In the case of `reverse_ordered_by`, the ordering will be such that for each successive element in the sequence, the first will be the element with the highest value of the first component in the structure ordering. Where ordering on multiple component is specified, the sequence will be sorted by the first component and then within each value of this, by the second component and so on.

A. Language Defined Characteristics

This appendix summarizes the definitions given elsewhere of the language-defined characteristics.

- `N'get_unique` for a name whose type is a bag, sequence or array types:
  `N'get_unique` returns a set containing all the elements of the collection; its type is the set type whose element type is the element of the collection.
- `N'elements` for a name `N`, whose type is a string, wstring, sequence or array type:
  `N'elements` returns a sequence containing all the elements of the value in the order defined by the value.
- `N'elements` for a name `N`, whose type is a set or bag type:
  `N'elements` returns a sequence containing all the elements of the value in an arbitrary order.
- `N'first` for a name `N`, whose type is a string, wstring, sequence or array type:
  `N'first` returns the index value of the first element of the value; its type is the corresponding index type. An exception is raised if the value is empty.
- `T'first` for a type `T`, that is an array type:
  `T'first` returns the first index of the type; its type is the corresponding index type.
- `T'first` for a type `T`, that is a numeric or enumeration type:
  `T'first` returns the first value that an entity of that type can take; its type is the type `T`.
- `N'image` for a name `N` whose type is a character, wcharacter, boolean, byte, numeric, structure, enumeration or collection type whose element type is one of the afore mentioned:
  `N'image` returns the string representation of the value; its type is string.
- `N'last` for a name `N` whose type is a string, wstring, sequence or array type:
  `N'last` returns the index value of the last element of the value; its type is the corresponding index type. An exception is raised if the value is empty.
- `T'last` for a type `T`, that is an array type:
  `T'last` returns the last index of the type; its type is the corresponding index type.

- `T'last` for a type `T`, that is a numeric or enumeration type:

  `T'last` returns the last value that an entity of that type can take; its type is the type `T`.

- `N'pred` for a name `N`, that is an enumeration type:

  `N'pred` returns the value whose position number is one less than that of the value; its type is the same type as `N`. An exception is raised if no such value exists.

- `N'range` for a name `N`, whose type is a string, wstring, sequence or array type:

  `N'range` returns the range of indices of the value; its type is the sequence type whose element type is the corresponding index type. The order of the elements in the range matches the order in the value.

- `T'range` for a type `T` that is an array type:

  `T'range` returns the range of indices of the type; its type is the sequence type whose element type is the corresponding index type. The order of the elements in the range matches the order given by the index type.

- `T'range` for the type `T` that is a numeric or enumeration type:

  `T'range` returns the range values that an entity of that type can take; its type is the sequence type whose element type is the type. The order of the elements in the range matches the order given by the type.

- `N'succ` for a name `N`, whose type is an enumeration type:

  `N'succ` returns the value whose position number is one more than that of the value; its type is the same type as `N`. An exception is raised if no such value exists.

- `N'length` for a name `N`, whose type is a string, wstring, set, bag, sequence or array type:

  `N'length` returns the length of the value, its type is integer.

- `N'upper` for a name `N`, whose type is a string or wstring type:

  `N'upper` returns an item of the same type with all the characters converted to upper case.

- `N'lower` for a name `N`, whose type is a string or wstring type:

  `N'lower` returns an item of the same type with all the characters converted to lower case.

- `N'firstcharpos` for a name `N`, whose type is a string or wstring type:

`N'firstcharpos(c)` returns an index to the first instance of the given character `c` in the name `N`.

- `T'value(s)` for a type `T` that is an enumeration type:

  `T'value(s)` returns the enumerate value corresponding to `s`.

B. Syntax Summary

This appendix gives a summary of grammatical syntax of MASL in EBNF [EBNF96]. The control codes are defined in ISO/IEC 6429:1992 [CCS92] as set C0.

[1]	goal	=	compilation unit ;
[2]	line terminator	=	? ISO 6429 Line Feed ? | ? ISO 6429 Carriage Return ? | ? ISO 6429 Carriage Return ?, ? ISO 6429 Line Feed ? ;
[3]	white space	=	' ? ISO 646 Space character ?' | ? ISO 6429 Horizontal Tabulation ? | ? ISO 6429 Form Feed ? | line terminator ;
[4]	single line comment	=	'//', {comment character}, line terminator ;
[5]	letter	=	'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j' | 'k' | 'l' | 'm' | 'n' | 'o' | 'p' | 'q' | 'r' | 's' | 't' | 'u' | 'v' | 'w' | 'x' | 'y' | 'z' | 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' | 'J' | 'K' | 'L' | 'M' | 'N' | 'O' | 'P' | 'Q' | 'R' | 'S' | 'T' | 'U' | 'V' | 'W' | 'X' | 'Y' | 'Z' ;
[6]	digit	=	'0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' ;
[7]	other character	=	'!' | '"' | '#' | '$' | '%' | '&' | ''' | '(' | ')' | '*' | '+' | ',' | '-' | '.' | '/' | ':' | ';' | '<' | '=' | '>' | '?' | '@' | '[' | '\' | ']' | '^' | '_' | '`' | '{' | '|' | '}' | '~' ;
[8]	comment character	=	letter | digit | other character | white space ;
[9]	identifier	=	letter {(letter | digit | underscore)} ;
[10]	underscore	=	'_' ;
[11]	relationship number	=	'R', non zero digit, {digit} ;
[12]	non zero digit	=	digit - '0' ;
[13]	reserved words	=	'abs' | 'and' | 'any' | 'array' | 'assigner' | 'associate' | 'bag' | 'begin' | 'boolean' | 'byte' | 'Cannot_Happen' | 'case' | 'character' | 'conditionally' | 'console' | 'create' | 'creation' | 'Current_State' | 'declare' | 'deferred' | 'delay' | 'delete' | 'delta' | 'device' | 'digits' | 'disunion' | 'domain' | 'elements' | 'else' | 'elsif' | 'end' | 'enum' | 'event' | 'exception' | 'exit' | 'false' | 'find_all' | 'find_one' | 'find_only' | 'find' | 'for' | 'from' | 'functions' | 'generate' | 'identifier' | 'if' | 'Ignore' | 'in' | 'instance' | 'intersection'

| 'is' | 'is_a' | 'link' | 'loop' | 'many' | 'mod' | 'native' | 'Non_Existent' | 'not' | 'not_in' | 'null' | 'numeric' | 'object' | 'of' | 'one' | 'ordered_by' | 'or' | 'others' | 'out' | 'pragma' | 'preferred' | 'private' | 'project' | 'public' | 'raises' | 'raise' | 'range' | 'readonly' | 'referential' | 'relationship' | 'rem' | 'return' | 'reverse_ordered_by' | 'reverse' | 'sequence' | 'service' | 'set' | 'start' | 'state' | 'string' | 'structure' | 'subtype' | 'terminal' | 'then' | 'this' | 'to' | 'transition' | 'true' | 'type' | 'unassociate' | 'unconditionally' | 'union' | 'unique' | 'unlink' | 'using' | 'wcharacter' | 'when' | 'while' | 'with' | 'wstring' | 'xor' ;

[14] literal = numeric literal | character literal | string literal | boolean literal | enumerator literal | null literal | device literal | stream literal ;

[15] numeric literal = decimal literal | based literal ;

[16] decimal literal = numeral, ('' | '.', numeral | exponent) ;

[17] numeral = digit, {digit} ;

[18] exponent = ('e'|'E'), [('+'|'-')], numeral ;

[19] based literal = base, '#', based numeral, ;

[20] base = numeral ;

[21] based numeral = extended digit, {extended digit} ;

[22] extended digit = digit | letter ;

[23] character literal = "'", single character, "'" | "'", escape sequence, "'" ;

[24] single character = [(letter | ((other character - "'") - '\')] ;

[25] string literal = '"', {string character}, '"' ;

[26] string character = [(letter | escape sequence, ((other character - '"') - '\')] ;

[27] escape sequence = '\n' | '\r' | '\t' | '\b' | '\f' | '\"' | '\'' | '\\' | octal escape | unicode escape ;

[28] octal escape = '\', octal lead digit, 2 * {octal digit} ;

[29] octal lead digit = '0' | '1' | '2' | '3' ;

[30] octal digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' ;

[31] unicode escape = '\u', extended digit, extended digit, extended digit, extended digit ;

[32]	boolean literal	=	'false' | 'true' ;
[33]	null literal	=	'null' ;
[34]	device literal	=	'console' ;
[35]	stream literal	=	'endl' | 'flush' ;
[36]	separator	=	'"' | '(' | ')' | '|' | ',' | '.' |'..' | ':' | '::' | ':=' | ';' | '=>' | '[' | ']' ;
[37]	operator	=	'*' | '**' | '+' | '-' | '&' | '->' | '/' | '/=' | '<' | '<=' | '=' | '>' | '>=' | '>>' | '<<' ;
[38]	pragma list	=	{pragma, ';'} ;
[39]	pragma	=	'pragma', pragma name, '(', pragma value list, ')' ;
[40]	pragma name	=	identifier ;
[41]	pragma value list	=	pragma value, {(',', pragma value)} ;
[42]	pragma value	=	identifier | literal ;
[43]	type	=	built-in type | collection type | instance type | user defined type ;
[44]	built-in type	=	'character' | 'wcharacter' | 'boolean' | 'byte' | 'string' | 'wstring' | 'device' | 'enum' | 'event' | 'instance' ;
[45]	collection type	=	set type | bag type | sequence type | array type ;
[46]	set type	=	'set of', type ;
[47]	bag type	=	'bag of', type ;
[48]	sequence type	=	'sequence', ['(', expression, ')'], 'of', type ;
[49]	array type	=	'array', '(', range, ')', 'of', type ;
[50]	instance type	=	'instance', 'of', scoped name ;
[51]	user defined type	=	scoped name ;
[52]	type declaration	=	type modifier, 'type', type name, 'is' type definition, ';' | type modifier, 'subtype', type name, 'is' subtype definition ;
[53]	type name	=	identifier ;
[54]	type modifier	=	[('public' | 'private')] ;

[55]	type definition	=	type, [constraint] | numeric type definition | structure type definition | enumeration type definition | unconstrained array type definition ;
[56]	numeric type definition	=	'numeric', constraint ;
[57]	constraint	=	range constraint | digits constraint | delta constraint ;
[58]	range constraint	=	'range', range ;
[59]	range	=	expression, '..', expression | name, '', 'range', [('(', ')')] | name, '', 'elements', [('(', ')')] | type, '', 'elements', [('(', ')')] | ;
[60]	digits constraint	=	'digits', constant expression, range constraint ;
[61]	delta constraint	=	'delta', constant expression, range constraint ;
[62]	structure type definition	=	'structure', component, {component} ;
[63]	component	=	component name, ':' type, [(':=', constant expression], ';' ;
[64]	component name	=	identifier ;
[65]	enumeration type definition	=	'enum', '(', enumerator, [(',', enumerator)], ')' ;
[66]	enumerator	=	enumerator name, ['=', constant expression] ;
[67]	enumerator name	=	identifier ;
[68]	enumerator literal	=	enumerator name | type name, '.', enumerator name | domain name, '::', enumerator name | domain name, '::', type name, '.', enumerator name ;
[69]	unconstrained array type definition	=	'array', '(', type, 'range', '<>', ')', 'of', type, ';' ;
[70]	subtype definition	=	user defined type, '(', range, ')', ';' | type, [constraint], ';' ;
[71]	variable declaration	=	variable name, ':', [modifier], type, [('(', range, ')')], [(':=', [extended expression], ';' ;
[72]	variable name	=	identifier ;

[73]	modifier	=	'readonly' ;
[74]	compilation unit	=	project declaration | domain declaration | domain definition ;
[75]	project declaration	=	'project', project name, 'is', {project declarative item}, 'end', ['project'], ';' ;
[76]	project name	=	identifier ;
[77]	project declaration item	=	'domain', domain name, ';' ;
[78]	domain declaration	=	'domain', domain name, 'is', {domain declarative item}, 'end', ['domain'], ';' ;
[79]	domain name	=	identifier, ;
[80]	domain declarative item	=	type declaration | exception declaration | object declaration | domain service declaration | domain function declaration | object predeclaration | relationship declaration ;
[81]	domain definition	=	state definition | object service definition | domain service definition | domain function definition | object function definition ;
[82]	scoped name	=	identifier | ('::', identifier) | (scoped name, '::', identifier) ;
[83]	object declaration	=	'object', object name, 'is', {object declarative item}, 'end', ['object'], ';' ;
[84]	object name	=	identifier ;
[86]	object declarative item	=	attribute declaration | object service declaration | object function declaration | event declaration | identifier declaration | state declaration | transition table declaration ;
[85]	object predeclaration	=	'object', object name, ';' ;
[87]	attribute declaration	=	attribute name, ':', attribute modifier, type, [('(', range , ')'], [(':=', constant expression)], ';' ;
[88]	attribute name	=	identifier ;
[89]	attribute modifier	=	['preferred'], [('referential', relationship formalization)] ;
[90]	identifier declaration	=	identifier, 'is', '(', attribute name, {(',', attribute name)}, ')', ';' ;

[91]	relationship formalization	=	'(', relationship specification, '.', attribute name, {(',', relationship specification, '.', attribute name)}, ')' ;
[92]	relationship declaration	=	'relationship', relationship number, 'is', relationship definition, ';' ;
[93]	relationship definition	=	subtype relationship definition | regular relationship definition ;
[94]	regular relationship definition	=	relationship description, ',', relationship description, [associative relationship] ;
[95]	relationship description	=	object name, relationship conditionality, role name, relationship multiplicity, object name ;
[96]	role name	=	identifier ;
[97]	relationship conditionality	=	'unconditionally' | 'conditionally' ;
[98]	relationship multiplicity	=	'one' | 'many' ;
[99]	associative relationship	=	'using', relationship multiplicity, object name ;
[100]	subtype relationship definition	=	object name, 'is_a', '(', object name, {(',', object name}, ')' ;
[101]	relationship specification	=	relationship number, '.', role, '.', object name | relationship number, '.', role | relationship number, '.', object name | relationship number ;
[102]	role	=	role name | 'is_a' ;
[103]	correlated relationship specification	=	relationship number, '.', role name, '.', object name | relationship number, '.', role name | relationship number, '.', object name | relationship number ;
[104]	state declaration	=	state modifier, 'state', state name, '(', parameter declaration list, ')', ';' ;
[105]	state name	=	identifier ;
[106]	state modifier	=	'assigner', ['start'] | 'creation' | 'terminal' ;
[107]	state definition	=	state modifier, 'state', domain name, '::', object name, '.', state name, '(', parameter declaration list, ')', 'is',

			{variable declaration, 'begin', handled sequence of statements, 'end', ['state'], ';' ;
[108]	event declaration	=	event modifier, 'event', event name, '(', parameter declaration list, ')', ';' ;
[109]	event name	=	identifier ;
[110]	event modifier	=	'assigner' | 'creation' ;
[111]	transition table declaration	=	transition modifier, 'transition', 'is', transition row, {transition row}, 'end', ['transition'], ';' ;
[112]	transition modifier	=	'assigner' ;
[113]	transition row	=	initial state, '(', transition, {(',', transition)}, ';' ;
[114]	initial state	=	'Non_Existent' | state name) ;
[115]	transition	=	incoming event, '=>', resultant state ;
[116]	incoming event	=	object name, '.', event name | event name ;
[117]	resultant state	=	'Ignore' | 'Cannot_Happen' | state name ;
[118]	domain function declaration	=	domain function modifier, 'function', function name, '(', function parameter declaration list, ')', 'return', type, [raises expression], ';' ;
[119]	function name	=	identifier ;
[120]	domain function modifier	=	'public' | 'private' ;
[121]	function parameter declaration list	=	function parameter declaration, {(',', function parameter declaration)} ;
[122]	function parameter declaration	=	function parameter name, ':', 'in', type ;
[123]	function parameter name	=	identifier ;
[124]	domain function definition	=	'function', domain name, '::', function name, '(', function parameter declaration list, ')', 'return', type, [raises expression], 'is', {variable declaration}, 'begin' handled sequence of statements, 'end', ['function'], ';' ;

[125]	object function declaration	=	object function modifier, 'function', function name, '(', function parameter declaration list, ')', 'return', type, [raises expression], ';' ;
[126]	object function modifier	=	('public' | 'private'), ['instance', ['deferred', '(', relationship number, ')']] ;
[127]	object function definition	=	object function modifier, 'function', domain name, '::', object name, '.' function name, '(', function parameter declaration list, ')', 'return', type, [raise expression], 'is', {variable declaration}, 'begin', handled sequence of statements, 'end', ['function'], ';' ;
[128]	domain service declaration	=	domain service modifier, 'service', service name, '(', parameter declaration list, ')', [raises expression], ';' ;
[129]	service name	=	identifier ;
[130]	domain service modifier	=	'public' | 'private' | 'native' ;
[131]	parameter declaration list	=	parameter declaration, {(',', parameter declaration)} ;
[132]	parameter declaration	=	parameter name, ':' parameter mode type ;
[133]	parameter name	=	identifier ;
[134]	parameter mode type	=	'in' | 'out' ;
[135]	raises expression	=	'raises', '(', exception specification, {exception specification}, ')' ;
[136]	domain service definition	=	('native', 'service', domain name, '::', service name, '(', parameter declaration list, ')', [raises expression], 'is', native code) | (domain service modifier, 'service', domain name, '::', service name, '(', parameter declaration list, ')', [raise expression], 'is', {variable declaration}, 'begin', handled sequence of statements, 'end', ['service'], ';' ;
[137]	native code	=	'$NATIVE', line terminator, {native code line}, '$ENDNATIVE', line terminator ;
[138]	native code line	=	{(letter | digit | other character | underscore | white space)}, line terminator ;

[139]	object service declaration	=	object service modifier, 'service', service name, '(', parameter declaration list, ')', [raise expression], ';' ;
[140]	object service modifier	=	('public' | 'private' | 'native'), ['instance', [('deferred', '(', relationship number, ')')]] ;
[141]	object service definition	=	('native', 'service', domain name, '::', object name, '.' service name, '(', parameter declaration list, ')', [raise expression], 'is', native code) | (object service modifier, 'service', domain name, '::', object name, '.' service name, '(', parameter declaration list, ')', [raises expression], 'is', {variable declaration}, 'begin', handled sequence of statements, 'end', ['service'], ';' ;
[142]	exception declaration	=	exception modifier, 'exception', exception name, ';' ;
[143]	exception name	=	identifier ;
[144]	exception modifier	=	'public' | 'private' ;
[145]	handled sequence of statements	=	sequence of statements, [('exception', {exception handler}, [exception others handler]] ;
[146]	exception handler	=	'when', exception choice, '=>', sequence of statements ;
[147]	exception others handler	=	'when', 'others', '=>', sequence of statements ;
[148]	exception choice	=	exception specification ;
[149]	exception specification	=	domain name, '::', exception name | exception name ;
[150]	sequence of statements	=	statement, {statement} ;
[151]	statement	=	simple statement | compound statement ;
[152]	simple statement	=	null statement | assignment statement | output stream | input stream | input line stream | invocation statement | exit statement | raise statement | delay statement | link statement | unlink statement | associate statement | unassociate statement | generate statement | pragma statement ;

[153]	compound statement	=	block statement | if statement | case statement | for statement | while statement ;
[154]	null statement	=	'null', ';' ;
[155]	assignment statement	=	{name}, ':=', extended expression, ';' ;
[156]	output statement	=	(name, '<<', expression, {('<<', expression}, ';') | device literal, '<<', expression, {('<<', expression}, ';') ;
[157]	input stream	=	(name, '>>', name, {('>>', name}, ';') | device literal, '>>', name, {('>>', name}, ';') ;
[158]	input line stream	=	(name, '>>>', name, {('>>>', name}, ';') | device literal, '>>>', name, {('>>>', name}, ';') ;
[159]	invocation statement	=	domain service invocation | object service invocation | instance service invocation ;
[160]	domain service invocation	=	(domain name, '::', service name, '(', argument list, ')', ';') | (service name, '(', argument list, ')', ';') ;
[161]	object service invocation	=	(object name, '.', service name, '(', argument list, ')', ';') | (instance expression, '.', service name, '(', argument list, ')', ';') | (service name, '(', argument list, ')', ';') ;
[162]	instance service invocation	=	(instance expression, '.', service name, '(', argument list, ')', ';') | (service name, '(', argument list, ')', ';') ;
[163]	argument list	=	(argument, {(',', argument} ;
[164]	argument	=	expression ;
[165]	block statement	=	['declare', variable declaration, {variable declaration}], 'begin', handled sequence of statements, 'end', ';' ;
[166]	if statement	=	'if', condition, 'then', sequence of statements, {('elsif', condition, 'then', sequence of statements)}, [('else', sequence of statements)], 'end', ['if'], ';' ;
[167]	case statement	=	'case', expression, 'is', {case statement alternative}, 'end', ['case'], ';' ;
[168]	case statement alternative	=	'when', discrete choice list, '=>', sequence of statements ;
[169]	discrete choice list	=	discrete choice, {'|', discrete choice list} ;
[170]	discrete choice	=	constant expression | 'others' ;

[171]	while statement	=	'while', condition, 'loop', sequence of statements, 'end', ['loop'], ';' ;
[172]	for statement	=	'for', loop parameter specification, 'loop', sequence of statements, 'end', ['loop'], ';' ;
[173]	loop parameter specification	=	variable name, 'in', ['reverse'], range ;
[174]	exit statement	=	'exit', [('when', condition)], ';' ;
[175]	raise statement	=	'raise', [exception specification], ';' ;
[176]	delete statement	=	('delete', name, ';') | ('delete', 'this', ';') ;
[177]	delay statement	=	'delay', expression ';' ;
[178]	link statement	=	'link', instance expression, relationship specification, instance expression, [('using', instance expression)], ';' ;
[179]	unlink statement	=	'unlink', instance expression, relationship specification, instance expression, [('using', instance expression)], ';' ;
[180]	associate statement	=	'associate', instance expression, relationship specification, instance expression, 'to', instance expression, ';' ;
[181]	unassociate statement	=	'unassociate', instance expression, relationship specification, instance expression, 'from', instance expression, ';' ;
[182]	generate statement	=	'generate', event specification, '(', argument list, ')', [('to', instance expression)], ';' ;
[183]	event specification	=	(object name, '.', event name) | event name ;
[184]	pragma statement	=	pragma, ';' ;
[185]	name	=	attribute name | parameter name | variable name | indexed component | selected component | selected attribute | slice ;
[186]	indexed component	=	prefix, '[', expression, ']' ;
[187]	prefix	=	name ;
[188]	selected component	=	prefix, '.', component name ;

[189]	selected attribute	=	prefix, '.', attribute name ;
[190]	slice	=	prefix, '[', range, ']' ;
[191]	primary expression	=	literal | 'this' | name as expression | characteristic reference | aggregate | service as expression | function invocation | event as expression | '(', expression, ')' ;
[196]	structure aggregate	=	'(', expression, ',', expression, {(',', expression)}, ')' ;
[193]	name as expression	=	name, ;
[194]	characteristic reference	=	name, '", characteristic reference, [('(', argument list, ')'] | type, '", characteristic reference, [('(', argument list ;
[195]	characteristic name	=	identifier ;
[192]	aggregate	=	structure aggregate | object aggregate ;
[197]	service as expression	=	domain name, '::', object name, '.', service name | domain name, '::', service name | object name, '.', service name | service name ;
[198]	event as expression	=	object name, '.', event name | event name, ;
[199]	function invocation	=	domain function invocation | object function invocation | instance function invocation ;
[200]	domain function invocation	=	domain name, '::', function name, '(', argument list, ')',';' | function name, '(', argument list, ')',';' ;
[201]	object function invocation	=	object name, '.', function name, '(', argument list, ')',';' | instance expression, '.', function name, '(', argument list, ')',';' | function name, '(', argument list, ')',';' ;
[202]	instance function invocation	=	instance expression, '.', function name, '(', argument list, ')',';' | function name, '(', argument list, ')',';' ;
[203]	unary expression	=	type conversion | '-', unary expression | '+', unary expression | 'not', unary expression | 'abs', unary expression | primary expression ;
[204]	type conversion	=	type, '(', expression, ')' ;

[205]	multiplicative expression	=	unary expression | multiplicative expression, '*', unary expression | multiplicative expression, '/', unary expression | multiplicative expression, 'mod', unary expression | multiplicative expression, '**', unary expression | multiplicative expression, 'rem', unary expression | multiplicative expression, 'intersection', unary expression | multiplicative expression, 'disunion', unary expression ;
[206]	additive expression	=	multiplicative expression | additive expression, '+', multiplicative expression | additive expression, '-', multiplicative expression | additive expression, '&', multiplicative expression | additive expression, 'union', multiplicative expression | additive expression, 'not_in', multiplicative expression ;
[207]	relational expression	=	additive expression | additive expression, '<', additive expression | additive expression, '>', additive expression | additive expression, '<=', additive expression | additive expression, '>=', additive expression ;
[208]	equality expression	=	relational expression | relational expression, '/=', relational expression | relational expression, '=', relational expression ;
[209]	logical and expression	=	equality expression | logical and expression, 'and', equality expression ;
[210]	logical xor expression	=	logical and expression | logical xor expression, 'xor', logical and expression ;
[211]	logical or expression	=	logical xor expression | logical or expression, 'or', logical xor expression ;
[212]	expression	=	logical or expression ;
[213]	condition	=	expression ;
[214]	instance expression	=	name | 'this' ;
[215]	instance collection expression	=	name ;

[216]	constant expression	=	expression ;
[217]	extended expression	=	create expression | find expression | relationship navigation | correlated relationship navigation | ordering expression | expression ;
[218]	create expression	=	'create', '.', ['unique'], object name, '(', object aggregate, ')' ;
[219]	object aggregate	=	attribute association, {(',', attribute association)} ;
[220]	attribute association	=	attribute name, '=>', expression | 'Current_State', '=>', state name ;
[221]	find expression	=	'find', find specification, '(', find condition, ')' | 'find_all', find specification, '(', ')' | 'find_one', find specification, '(', [find condition], ')' | 'find_only', find specification, '(', [find condition], ')' ;
[222]	find specification	=	scoped object name | instance collection expression ;
[223]	scoped object name	=	scoped name ;
[224]	find condition	=	find xor condition | find condition, 'or', find xor condition ;
[225]	find xor condition	=	find and condition | find xor condition, 'xor', find and condition ;
[226]	find and condition	=	find equality condition | find and condition, 'and', find equality condition ;
[227]	find equality condition	=	find relational condition | name, '/=', relational expression | name, '=', relational expression ;
[228]	find relational condition	=	find unary condition | name, '<', additive expression | name, '>', additive expression | name, '<=', additive expression | name, '>=', additive expression ;
[229]	find unary condition	=	'not', find unary condition | '(', find condition, ')' ;
[230]	relationship navigation	=	instance expression, '->', relationship specification, {('->', relationship specification)} | instance collection expression, '->', relationship specification, {('->', relationship specification)} ;

[231]	correlated relationship navigation	=	instance expression, 'with', instance expression, '->', relationship specification ;
[232]	ordering expression	=	instance ordering expression | structure ordering expression ;
[233]	instance ordering expression	=	instance collection expression, 'ordered_by', '(', instance ordering, ')' | instance collection expression, 'reverse ordered_by', '(', instance ordering, ')' | find expression, 'ordered_by', '(', instance ordering, ')' | find expression, 'reverse_ordered_by', '(', instance ordering, ')' ;
[234]	instance ordering	=	attribute name, {('', attribute name)} ;
[235]	structure ordering expression	=	prefix, 'ordered_by','(', structure ordering, ')' | prefix, 'reverse_ordered_by','(', structure ordering, ')' ;
[236]	structure ordering	=	component name, {(',', component name)} ;

Index

Symbols

A

B

C

D

E

F

P

R

S

T