ISO TC184/SC4/* WG5 N 65 (P 2)

*Complete with EC (for Editing), PMAG, or WG

Date: 1 November 1994 **Supercedes SC4/ WG5 N 55/60(P 2)**

PRODUCT DATA REPRESENTATION AND EXCHANGE

Part: 11 **Title:** EXPRESS Language Reference Manual

Purpose of this document as it relates to the target document is:

X Primary Content
___ Issue Discussion
___ Alternate Proposal
___ Partial Content

Current Status: ISO IS

ABSTRACT:
This document contains the definition of the EXPRESS lexical information modelling language, together with the definition of the EXPRESS-G iconic language which is a subset of EXPRESS.

KEYWORDS:
EXPRESS, EXPRESS-G, Information modelling

Document Status/Dates (dd/mm/yy)

Part Documents		Other SC4 Documents
28/2/91	Released	___ Released
8/2/91	Project	___ Working
9/12/90	Working	___ Editorial OK
11/4/91	Technically Complete	___ Technically
11/4/91	Editorially Complete	Complete
24/10/91	ISO Committee Draft	___ Approved

Owner/Editor: Philip Spiby
Address: CADDETC
Arndale House
Headingley
Leeds LS6 2UU
United Kingdom
Telephone/FAX: +44 532 305005
E-mail: philsp@leva.icf.leeds.ac.uk

Alternate: Doug Schenck
Address: McDonnell Aircraft Co
Dept 470
Bldg 271D, MS 2761360
PO Box 516
St Louis, MO 63166
Telephone/FAX: +1 314 2345258
E-mail:

Comments to Reader

This document contains the resolutions agreed by the EXPRESS Language Project (WG5/P3) and agreed by PMAG to the ISO ballot comments on the previous document when issued as an ISO DIS. It also incorporates the editorial comments from the ISO, Geneva review in September 1994.

Contents

Annexes

Figures

Tables

Foreword

The International Organization for Standardization (ISO) is a worldwide federation of national standards bodies (ISO member bodies). The work of preparing International Standards is normally carried out through ISO technical committees. Each member body interested in a subject for which a technical committee has been established has the right to be represented on that committee. International organizations, governmental and non-governmental, in liaison with ISO, also take part in the work. ISO collaborates closely with the International Electrotechnical Commission (IEC) on all matters of electrotechnical standardization.

Draft International Standards adopted by technical committees are circulated to the member bodies for voting. Publication as an International Standard requires approval by at least 75% of the member bodies casting a vote.

International Standard ISO 10303–11 was prepared by Technical Committee ISO/TC 184, *Industrial automation systems and integration,* Subcommittee SC4, *Industrial data and global manufacturing programming languages.*

This part of ISO 10303 is based in part upon material in:

– ISO 10646-1:1993, Information technology – Universal multiple-octet coded character set (UCS) – Architecture and basic multilingual plane.

ISO 10303 consists of the following parts under the general title *Industrial automation systems and integration – Product data representation and exchange:*

– Part 1, Overview and fundamental principles;

– Part 11, Description methods: The *EXPRESS* language reference manual;

– Part 21, Implementation methods: Clear text encoding of the exchange structure;

– Part 22, Implementation methods: Standard data access interface specification;

– Part 31, Conformance testing methodology and framework: General concepts;

– Part 32, Conformance testing methodology and framework: Requirements on testing laboratories and clients;

– Part 41, Integrated generic resources: Fundamentals of product description and support;

– Part 42, Integrated generic resources: Geometric and topological representation;

– Part 43, Integrated generic resources: Representation structures;

– Part 44, Integrated generic resources: Product structure configuration;

– Part 45, Integrated generic resources: Materials;

– Part 46, Integrated generic resources: Visual presentation;

- Part 47, Integrated generic resources: Shape variation tolerances;
- Part 49, Integrated generic resources: Process structure and properties;
- Part 101, Integrated application resources: Draughting;
- Part 104, Integrated application resources: Finite element analysis;
- Part 105, Integrated application resources: Kinematics;
- Part 201, Application protocol: Explicit draughting;
- Part 202, Application protocol: Associative draughting;
- Part 203, Application protocol: Configuration controlled design;
- Part 207, Application protocol: Sheet metal die planning and design;
- Part 210, Application protocol: Printed circuit assembly product design data;
- Part 213, Application protocol: Numerical control process plans for machined parts.

The structure of this International Standard is described in ISO 10303-1. The numbering of the parts of this International Standard reflects its structure:

- Part 11 specifies the description methods;
- Parts 21 and 22 specify the implementation methods;
- Parts 31 and 32 specify the conformance testing methodology and framework;
- Parts 41 to 49 specify the integrated generic resources;
- Parts 101 to 105 specify the integrated application resources;
- Parts 201 to 213 specify the application protocols.

Should further parts be published, they will follow the same numbering pattern.

Annexes A, B, C, D, E and F form an integral part of this part of ISO 10303. Annexes G, H and J are for information only.

Introduction

ISO 10303 is an International Standard for the computer-interpretable representation and exchange of product data. The objective is to provide a neutral mechanism capable of describing product data throughout the life cycle of a product independent from any particular system. The nature of this description makes it suitable not only for neutral file exchange, but also as a basis for implementing and sharing product databases and archiving.

This International Standard is organized as a series of parts, each published separately. The parts of ISO 10303 fall into one of the following series: description methods, integrated resources, application protocols, abstract test suites, implementation methods, and conformance testing. The series are described in ISO 10303–1. This part of ISO 10303 is a member of the descriptive methods series.

This part of ISO 10303 specifies the elements of the *EXPRESS* language. Each element of the language is presented in its own context with examples. Simple elements are introduced first, then more complex ideas are presented in an incremental manner.

Language overview

EXPRESS is the name of a formal information requirements specification language. It is used to specify the information requirements of other parts of this International Standard. It is based on a number of design goals among which are:

– the size and complexity of ISO 10303 demands that the language be parsable by both computers and humans. Expressing the information elements of ISO 10303 in a less formal manner would eliminate the possibility of employing computer automation in checking for inconsistencies in presentation or for creating any number of secondary views, including implementation views;

– the language is designed to enable partitioning of the diverse material addressed by ISO 10303. The schema is the basis for partitioning and intercommunication;

– the language focuses on the definition of entities, which represent objects of interest. The definition of an entity is in terms of its properties, which are characterized by specification of a domain and the constraints on that domain;

– the language seeks to avoid, as far as possible, specific implementation views. However, it is possible to manufacture implementation views (such as static file exchange) in an automatic and straightforward manner.

In *EXPRESS*, entities are defined in terms of attributes: the traits or characteristics considered important for use and understanding. These attributes have a representation which might be a simple data type (such as integer) or another entity type. A geometric point might be defined in terms of three real numbers. Names are given to the attributes which contribute to the definition of an entity. Thus, for a geometric point the three real numbers might be named x, y and z. A relationship is established between the entity being defined and the attributes that define it, and, in a similar manner, between the attribute and its representation.

NOTES

1 – A number of languages have contributed to *EXPRESS*, in particular, Ada, Algol, C, C++, Euler, Modula-2, Pascal, PL/I and SQL. Some facilities have been invented to make *EXPRESS* more suitable for the job of expressing an information model.

2 – The examples of *EXPRESS* usage in this manual do not conform to any particular style rules. Indeed, the examples sometimes use poor style to conserve space or to show flexibility. The examples are not intended to reflect the content of the information models defined in other parts of this International Standard. They are crafted to show particular features of *EXPRESS*. Any similarity between the examples and the normative information models specified in other parts of ISO 10303 should be ignored.

Industrial automation systems and integration — Product data representation and exchange — Part 11 : Description methods: The EXPRESS language reference manual

1 Scope

This part of ISO 10303 defines a language by which aspects of product data can be specified. The language is called *EXPRESS*.

This part of ISO 10303 also defines a graphical representation for a subset of the constructs in the *EXPRESS* language. This graphical representation is called *EXPRESS-G*.

EXPRESS is a data specification language as defined in ISO 10303-1. It consists of language elements which allow an unambiguous data definition and specification of constraints on the data defined.

The following are within the scope:

- data types;
- constraints on instances of the data types.

The following are outside the scope of this part of ISO 10303:

- definition of database formats;
- definition of file formats;
- definition of transfer formats;
- process control;
- information processing;
- exception handling.

EXPRESS is not a programming language.

2 Normative references

The following standards contain provisions which, through reference in this text, constitute provisions of this part of ISO 10303. At the time of publication, the editions indicated were

valid. All standards are subject to revision, and parties to agreements based on this part of ISO 10303 are encouraged to investigate the possibility of applying the most recent editions of the standards indicated below. Members of IEC and ISO maintain registers of currently valid International Standards.

ISO 10303-1:1994, *Industrial automation systems and integration – Product data representation and exchange – Part 1: Overview and fundamental principles*

ISO/IEC 8824-1:—[1)], *Information technology – Open systems interconnection – Abstract syntax notation one (ASN.1) – Part 1: Specification of basic notation.*

ISO/IEC 10646-1:1993, *Information technology – Universal multiple-octet coded character set (UCS) – Part 1: Architecture and basic multilingual plane.*

3 Definitions

3.1 Terms defined in ISO 10303-1

This part of ISO 10303 makes use of the following terms defined in ISO 10303-1:

- Context;
- Data;
- Data specification language;
- Information.

3.2 Other definitions

For the purposes of this part of ISO 10303, the following definitions apply:

3.2.1 complex entity data type: a representation of an entity. A complex entity data type establishes a domain of values defined by the common attributes and constraints of an allowed combination of entity data types within a particular subtype/supertype graph.

3.2.2 complex entity (data type) instance: a named unit of data which represents a unit of information within the class defined by an entity. It is a member of the domain established by a complex entity data type.

3.2.3 constant: a named unit of data from a specified domain. The value cannot be modified.

3.2.4 data type: a domain of values.

[1)] To be published.

3.2.5 entity: a class of information defined by common properties.

3.2.6 entity data type: a representation of an entity. An entity data type establishes a domain of values defined by common attributes and constraints.

3.2.7 entity (data type) instance: a named unit of data which represents a unit of information within the class defined by an entity. It is a member of the domain established by an entity data type.

3.2.8 instance: a named value.

3.2.9 partial complex entity data type: a potential representation of an entity. A partial complex entity data type is a grouping of entity data types within a subtype/supertype graph which may form part or all of a complex entity data type.

3.2.10 partial complex entity value: a value of a partial complex entity data type. This has no meaning on its own and must be combined with other partial complex entity values and a name to form a complex entity instance.

3.2.11 population: a collection of entity data type instances.

3.2.12 simple entity (data type) instance: a named unit of data which represents a unit of information within the class defined by an entity. It is a member of the domain established by a single entity data type.

3.2.13 subtype/supertype graph: a declared collection of entity data types. The entity data types declared within a subtype/supertype graph are related via the subtype statement. A subtype/supertype graph defines one or more complex entity data types.

3.2.14 token: a non-decomposable lexical element of a language.

3.2.15 value: a unit of data.

4 Conformance requirements

4.1 Formal specifications written in EXPRESS

4.1.1 Lexical language

A formal specification written in *EXPRESS* shall be consistent with a given level as specified below. A formal specification is consistent with a given level when all checks identified for that level and all lower levels are verified for the specification.

Levels of checking

Level 1: Reference checking. This level consists of checking the formal specification to ensure that it is syntactically and referentially valid. A formal specification is syntactically valid if it matches the syntax generated by expanding the primary syntax rule (`syntax`) given in annex A. A formal specification is referentially valid if all references to *EXPRESS* items are consistent with the scope and visibility rules defined in 10 and 11.

Level 2: Type checking. This level consists of checking the formal specification to ensure that it is consistent with the following:

- expressions shall comply with the rules specified in clause 12;
- assignments shall comply with the rules specified in 13.3;
- inverse attribute declarations shall comply with the rules specified in 9.2.1.3;
- attribute redeclarations shall comply with the rules specified in 9.2.3.4.

Level 3: Value checking. This level consists of checking the formal specification to ensure that it complies with statements of the form 'A shall be greater than B' as specified in clause 7 to clause 16. This is limited to those places where both A and B can be evaluated from literals and/or constants.

Level 4: Complete checking. This level consists of checking a formal specification to ensure that it complies with all statements of requirement as specified in this part of ISO 10303.

EXAMPLE 1 – This part of ISO 10303 states that functions shall specify a return statement in each of the possible paths a process may take when that function is invoked. This would have to be checked.

4.1.2 Graphical form

A formal specification written in *EXPRESS-G* shall be consistent with a given level as specified below. A formal specification is consistent with a given level when all checks identified for that level and all lower levels are verified for the specification.

Levels of checking

Level 1: Symbols and scope checking. This level consists of checking the formal specification to ensure that it is consistent with either an entity level or a schema level specification as defined in D.5 and D.6 respectively. This includes checking that the formal specification uses symbols as defined in D.2, D.3 and D.4. The formal specification will also be checked to ensure that page references and redeclared attributes comply with D.4.1 and D.5.5 respectively.

Level 2: Complete checking. This level consists of checking a formal specification to identify those places which do not conform to either a complete entity level or complete schema level specification as defined in annex D and the requirements defined in clause 7 through clause 16.

4.2 Implementations of EXPRESS

4.2.1 EXPRESS language parser

An implementation of an *EXPRESS* language parser shall be able to parse any formal specification written in *EXPRESS*, consistent with the constraints as specified in the annex E associated with that implementation. An *EXPRESS* language parser shall be said to conform to a particular checking level (as defined in 4.1.1) if it can apply all checks required by the level (and any level below that) to a formal specification written in *EXPRESS*.

The implementor of an *EXPRESS* language parser shall state any constraints which the implementation imposes on the number and length of identifiers, on the range of processed numbers, and on the maximum precision of real numbers. Such constraints shall be documented in the form specified by annex E for the purposes of conformance testing.

4.2.2 Graphical editing tool

An implementation of an *EXPRESS-G* editing tool shall be able to create and display a formal specification in *EXPRESS-G*, consistent with the constraints as specified in the annex E associated with that implementation. An *EXPRESS-G* editing tool shall be said to conform to a particular checking level if it can create and display a formal specification in *EXPRESS-G* which is consistent with the specified level of checking (and any level below that).

The implementor of an *EXPRESS-G* editing tool shall state any constraints which the implementation imposes on the number and length of identifiers, the number of symbols available per page of the model, and the maximum number of pages. Such constraints shall be documented in the form specified by annex E for the purposes of conformance testing.

5 Fundamental principles

The reader of this document is assumed to be familiar with the following concepts.

A schema written in the *EXPRESS* language describes a set of conditions which establishes a

domain. Instances can be evaluated to determine if they are in the domain. If the instances meet all the conditions, then they are asserted to be in the domain. If the instances fail to meet any of the conditions, then the instances have violated the conditions and thus are not in the domain. In the case where the instances do not contain values for optional attributes and some of the conditions involve those optional attributes, it may not be possible to determine whether the instances meet all the conditions. In this case, the instances are considered in the domain.

Many of the elements of the *EXPRESS* language are assigned names. The name allows other language elements to reference the associated representation. The use of the name in the definition of other language elements constitutes a reference to the underlying representation. While the syntax of the language uses an identifier for the name, the underlying representation must be examined to understand the structure.

The specification of an entity data type in the *EXPRESS* language describes a domain. The individual members of the domain are assumed to be distinguished by some associated identifier which is unique. *EXPRESS* does not specify the content or representation of these identifiers.

The declaration of a constant entity instance defines an identifiable member of the domain described by the entity data type. These entity instances shall not be modified or deleted by operations performed on the domain.

The procedural description of constraints in *EXPRESS* may declare or make reference to additional entity instances, as local variables, which are assumed to be transient identifiable members of the domain. These procedural descriptions may modify these additional entity instances, but cannot modify persistent members of the domain. These transient members of the domain are only accessible within the scope of the procedural code in which they were declared, and cease to exist upon termination of that code.

The *EXPRESS* language does not describe an implementation environment. In particular, *EXPRESS* does not specify:

- how references to names are resolved;
- how other schemas are known;
- how or when constraints are checked;
- what an implementation shall do if a constraint is not met;
- whether or not instances that do not conform to an *EXPRESS* schema are allowed to exist in an implementation;
- whether, when or how instances are created, modified or deleted in an implementation.

6 Language specification syntax

The notation used to present the syntax of the *EXPRESS* language is defined in this clause.

The full syntax for the *EXPRESS* language is given in annex A. Portions of those syntax rules are

reproduced in various clauses to illustrate the syntax of a particular statement. Those portions are not always complete. It will sometimes be necessary to consult annex A for the missing rules. The syntax portions within this part of ISO 10303 are presented in a box. Each rule within the syntax box has a unique number toward the left margin for use in cross references to other syntax rules.

6.1 The syntax of the specification

The syntax of *EXPRESS* is defined in a derivative of Wirth Syntax Notation (WSN).

NOTE – See annex J under [3] for a reference.

The notational conventions and WSN defined in itself are given below.

```
syntax          = { production } .
production      = identifier '=' expression '.' .
expression      = term { '|' term } .
term            = factor { factor } .
factor          = identifier | literal | group | option | repetition .
identifier      = character { character } .
literal         = '''' character { character } '''' .
group           = '(' expression ')' .
option          = '[' expression ']' .
repetition      = '{' expression '}' .
```

– The equal sign '=' indicates a production. The element on the left is defined to be the combination of the elements on the right. Any spaces appearing between the elements of a production are meaningless unless they appear within a literal. A production is terminated by a period '.'.

– The use of an identifier within a factor denotes a nonterminal symbol which appears on the left side of another production. An identifier is composed of letters, digits and the underscore character. The keywords of the language are represented by productions whose identifier is given in uppercase characters only.

– The word literal is used to denote a terminal symbol which cannot be expanded further. A literal is a case independent sequence of characters enclosed in apostrophes. Character, in this case, stands for any character as defined by ISO 10646 cells 21-7E in group 00, plane 00, row 00. For an apostrophe to appear in a literal it must be written twice.

– The semantics of the enclosing braces are defined below:

- curly brackets '{ }' indicates zero or more repetitions;
- square brackets '[ ]' indicates optional parameters;
- parenthesis '()' indicates that the group of productions enclosed by parenthesis shall be used as a single production;

- vertical bar '|' indicates that exactly one of the terms in the expression shall be chosen.

EXAMPLE 2 – The syntax for a string type is as follows:

Syntax:

```
293 string_type = STRING [ width_spec ] .
318 width_spec = '(' width ')' [ FIXED ] .
317 width = numeric_expression .
```

The complete syntax definition (annex A) contains the definitions for `STRING`, `numeric_expression` and `FIXED`.

EXAMPLE 3 – Following the syntax given in example 2, the following alternatives are possible:

a) `string`

b) `string ( 22 )`

c) `string ( 19 ) fixed`

The rule for `numeric_expression` is quite complex and allows many other things to be written.

6.2 Special character notation

The following notation is used to represent entire character sets and certain special characters which are difficult to display:

– `\a` represents characters in cells 21-7E of row 00, plane 00, group 00 of ISO 10646;

– `\n` represents a newline (system dependent) (see 7.1.5.2);

– `\q` is the quote (apostrophe) (') character and is contained within `\a`;

– `\s` is the space character;

– `\o` represents characters in cells 00-1F and 7F of row 00, plane 00, group 00 of ISO 10646.

7 Basic language elements

This clause specifies the basic elements from which an *EXPRESS* schema is composed: the character set, remarks, symbols, reserved words, identifiers and literals.

The basic language elements are composed into a stream of text, typically broken into physical lines. A physical line is any number (including zero) of characters ended by a newline (see

7.1.5.2).

NOTE – A schema is easier to read when statements are broken into lines and whitespace is used to layout the different constructs.

EXAMPLE 4 – The following are equivalent

```
entity point;x,y,z:real;end_entity;

ENTITY point;
  x,
  y,
  z : REAL;
END_ENTITY;
```

7.1 Character set

A schema written in *EXPRESS* shall only use the characters defined by the following selected subset of ISO 10646: cells 00 to 7F of row 00 of plane 00 of group 00. This selected subset of ISO 10646 is called the *EXPRESS* character set. Members of this set are referred to by the cell of ISO 10646 in which these characters are defined, these cell numbers are specified in hexadecimal. The printable characters from this subset (cells 21 - 7E) are combined to form the tokens for the *EXPRESS* language. The *EXPRESS* tokens are keywords, identifiers, symbols or literals. The *EXPRESS* character set is further classified below:

NOTE – This clause only refers to the characters used to specify an *EXPRESS* schema, and does not specify the domain of characters allowed within a string data type.

7.1.1 Digits

EXPRESS uses the Arabic digits 0-9 (cells 30 - 39 of the *EXPRESS* character set).

Syntax:

```
120 digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' .
```

7.1.2 Letters

EXPRESS uses the upper and lower case letters of the English alphabet (cells 41 - 5A and 61 - 7A of the *EXPRESS* character set). The case of letters is significant only within explicit string literals.

NOTE – *EXPRESS* may be written using upper, lower or mixed case letters (see example 4).

Syntax:

```
124 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' .
```

7.1.3 Special characters

The special characters (printable characters which are neither letters nor digits) are used mainly for punctuation and as operators. The special characters are in cells 21-2F, 3A-3F, 40, 5B-5E, 60 and 7B-7E of the *EXPRESS* character set.

Syntax:

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

7.1.4 Underscore

The underscore character (_, cell 5F of the *EXPRESS* character set) may be used in identifiers and keywords, with the exception that the underscore character cannot be used as the first character.

7.1.5 Whitespace

Whitespace is defined by the following sub-clauses and by 7.1.6. Whitespace shall be used to separate the tokens of a schema written in *EXPRESS*.

NOTE – Liberal, and consistent, use of whitespace can improve the structure and readability of a schema.

7.1.5.1 Space character

One or more spaces (cell 20 of the *EXPRESS* character set) can appear between two tokens or within an explicit string literal. The notation `\s` is used to represent a blank space character in the syntax of the language.

7.1.5.2 Newline

A newline marks the physical end of a line within a formal specification written in *EXPRESS*. Newline is normally treated as a space but is significant when it terminates a tail remark or abnormally terminates a string literal. A newline is represented by the notation `\n` in the syntax of the language.

The representation of a newline is implementation specific.

7.1.5.3 Other characters

Characters not defined in 7.1.1 to 7.1.5.2 (i.e., cells 00 - 1F and 7F of the *EXPRESS* character set) shall be treated as whitespace, unless within an explicit string literal. The notation `\o` shall

be used to represent any one of these other characters in the syntax of the language.

7.1.6 Remarks

A remark is used for documentation and shall be interpreted by an *EXPRESS* language parser as whitespace. There are two forms of remark, embedded remark and tail remark.

7.1.6.1 Embedded remark

The character pair (* denotes the start of an embedded remark and the character pair *) denotes its end. An embedded remark may appear between any two tokens.

Syntax:

```
142 embedded_remark = '(*' { not_lparen_star | lparen_not_star | star_not_rparen |
                      embedded_remark } '*)' .
126 not_lparen_star = not_paren_star | ')' .
127 not_paren_star = letter | digit | not_paren_star_special .
124 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' .
120 digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' .
129 not_paren_star_special = not_paren_star_quote_special | '''' .
128 not_paren_star_quote_special = '!' | '"' | '#' | '$' | '%' | '&' | '+' | ',' |
                                   '-' | '.' | '/' | ':' | ';' | '<' | '=' | '>' |
                                   '?' | '@' | '[' | '\' | ']' | '^' | '_' | '`' |
                                   '{' | '|' | '}' | '~' .
125 lparen_not_star = '(' not_star .
132 not_star = not_paren_star | '(' | ')' .
135 star_not_rparen = '*' not_rparen .
131 not_rparen = not_paren_star | '*' | '(' .
```

Any character within the *EXPRESS* character set may occur between the start and end of an embedded remark including the newline character; therefore embedded remarks can span several physical lines.

Embedded remarks may be nested.

NOTE – Care must be taken when nesting remarks to ensure that there are matched pairs of symbols.

EXAMPLE 5 – The following is an example of nested embedded remarks.

```
(* The '(*' symbol starts a remark, and the '*)' symbol ends it *)
```

7.1.6.2 Tail remark

The tail remark is written at the end of a physical line. Two consecutive hyphens (--) start the tail remark and the following newline terminates it.

Syntax:

```
144 tail_remark = '--' { \a | \s | \o } \n .
```

EXAMPLE 6 – `-- this is a remark that ends with a newline`

7.2 Reserved words

The reserved words of *EXPRESS* are the keywords and the names of built-in constants, functions and procedures. The reserved words shall not be used as identifiers. The reserved words of *EXPRESS* are described below.

7.2.1 Keywords

The *EXPRESS* keywords are shown in table 1.

NOTES

1 – Keywords have an uppercase production which represents the literal. This is to enable easier reading of the syntax productions.

2 – CONTEXT, END_CONTEXT, MODEL and END_MODEL are reserved for use in later editions of this part of this International Standard.

Table 1 – EXPRESS keywords

ABSTRACT	AGGREGATE	ALIAS	ARRAY
AS	BAG	BEGIN	BINARY
BOOLEAN	BY	CASE	CONSTANT
CONTEXT	DERIVE	ELSE	END
END_ALIAS	END_CASE	END_CONSTANT	END_CONTEXT
END_ENTITY	END_FUNCTION	END_IF	END_LOCAL
END_MODEL	END_PROCEDURE	END_REPEAT	END_RULE
END_SCHEMA	END_TYPE	ENTITY	ENUMERATION
ESCAPE	FIXED	FOR	FROM
FUNCTION	GENERIC	IF	INTEGER
INVERSE	LIST	LOCAL	LOGICAL
MODEL	NUMBER	OF	ONEOF
OPTIONAL	OTHERWISE	PROCEDURE	QUERY
REAL	REFERENCE	REPEAT	RETURN
RULE	SCHEMA	SELECT	SET
SKIP	STRING	SUBTYPE	SUPERTYPE
THEN	TO	TYPE	UNIQUE
UNTIL	USE	VAR	WHERE
WHILE			

7.2.2 Reserved words which are operators

The operators defined by reserved words are shown in table 2. See clause 12 for the definition of these operators.

Table 2 – EXPRESS reserved words which are operators

AND	ANDOR	DIV	IN
LIKE	MOD	NOT	OR
XOR			

7.2.3 Built-in constants

The names of the built-in constants are shown in table 3. See clause 14 for the definitions of these constants.

Table 3 – EXPRESS reserved words which are constants

?	SELF	CONST_E	PI
FALSE	TRUE	UNKNOWN	

7.2.4 Built-in functions

The names of the built-in functions are shown in table 4. See clause 15 for the definitions of these functions.

Table 4 – EXPRESS reserved words which are function names

ABS	ACOS	ASIN	ATAN
BLENGTH	COS	EXISTS	EXP
FORMAT	HIBOUND	HIINDEX	LENGTH
LOBOUND	LOG	LOG2	LOG10
LOINDEX	NVL	ODD	ROLESOF
SIN	SIZEOF	SQRT	TAN
TYPEOF	USEDIN	VALUE	VALUE_IN
VALUE_UNIQUE			

7.2.5 Built-in procedures

The names of the built-in procedures are shown in table 5. See clause 16 for the definitions of these procedures.

Table 5 – EXPRESS reserved words which are procedure names

INSERT	REMOVE

7.3 Symbols

Symbols are special characters or groups of special characters which have special meaning in *EXPRESS*. Symbols are used in *EXPRESS* as delimiters and operators. A delimiter is used to begin, separate or terminate adjacent lexical or syntactic elements. Interpretation of these elements would be impossible without separators. Operators denote that actions shall be performed on the operands which are associated with the operator; see clause 12 for an explanation of operators. The *EXPRESS* symbols are shown in table 6.

Table 6 – EXPRESS symbols

.	,	;	:
*	+	-	=
%	'	\	/
<	>	[	]
{	}	|	e
(	)	<=	<>
>=	<*	:=	||
**	--	(*	*)
:=:	:<>:		

7.4 Identifiers

Identifiers are names given to the items declared in a schema (see 9.3), including the schema itself. An identifier shall not be the same as an *EXPRESS* reserved word.

Syntax:

```
140 simple_id = letter { letter | digit | '_' } .
124 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' .
120 digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' .
```

The first character of an identifier shall be a letter. The remaining characters, if any, may be any combination of letters, digits and the underscore character.

The implementor of an *EXPRESS* language parser shall specify the maximum number of characters of an identifier which can be read by that implementation, using annex E.

7.5 Literals

A literal is a self-defining constant value. The type of a literal depends on how characters are composed to form a token. The literal types are binary, integer, real, string and logical.

Syntax:

```
238 literal = binary_literal | integer_literal | logical_literal | real_literal |
              string_literal .
```

7.5.1 Binary literal

A binary literal represents a value of a binary data type and is composed of the % symbol followed by one or more bits (0 or 1).

Syntax:

```
136 binary_literal = '%' bit { bit } .
119 bit = '0' | '1' .
```

The implementor of an *EXPRESS* language parser shall specify the maximum number of bits in a binary literal which can be read by that implementation, using annex E.

EXAMPLE 7 – A valid binary literal

```
%0101001100
```

7.5.2 Integer literal

An integer literal represents a value of an integer data type and is composed of one or more digits.

Syntax:

```
138 integer_literal = digits .
121 digits = digit { digit } .
120 digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' .
```

NOTE – The sign of the integer literal is not modelled within the syntax since *EXPRESS* uses the concept of unary operators within the expression syntax.

The implementor of an *EXPRESS* language parser shall specify the maximum integer value of an integer literal which can be read by that implementation, using annex E.

EXAMPLE 8 – Valid integer literals

```
4016
38
```

7.5.3 Real literal

A real literal represents a value of a real data type and is composed of a mantissa and an optional exponent; the mantissa shall include a decimal point.

Syntax:

```
139 real_literal = digits '.' [ digits ] [ 'e' [ sign ] digits ] .
121 digits = digit { digit } .
120 digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' .
286 sign = '+' | '-' .
```

NOTE – The sign of the real literal is not modelled within the syntax since *EXPRESS* uses the concept of unary operators within the expression syntax.

The implementor of an *EXPRESS* language parser shall specify the maximum precision and maximum exponent of a real literal which can be read by that implementation, using annex E.

EXAMPLES

9 – Valid real literals

```
1.E6        "E" may be written in upper or lower case
3.5e-5
359.62
```

10 – Invalid real literals

```
.001    At least one digit must precede the decimal point
1e10    A decimal point must be part of the literal
1. e10  A space is not part of the real literal
```

7.5.4 String literal

A string literal represents a value of a string data type. There are two forms of string literal, the simple string literal and encoded string literal. A simple string literal is composed of a sequence of characters in the *EXPRESS* character set (see 7.1) enclosed by quote marks (’). A quote mark within a simple string literal is represented by two consecutive quote marks. An encoded string literal is a four octet encoded representation of each character in a sequence of ISO 10646 characters enclosed in double quote marks ("). The encoding is defined as follows:

– first octet = ISO 10646 group in which the character is defined;

– second octet = ISO 10646 plane in which the character is defined;

– third octet = ISO 10646 row in which the character is defined;

– fourth octet = ISO 10646 cell in which the character is defined.

A string literal shall never span a physical line boundary; that is, a newline shall not occur between the quotes enclosing a string literal.

Syntax:

```
292 string_literal = simple_string_literal | encoded_string_literal .
141 simple_string_literal = \q { ( \q \q ) | not_quote | \s | \o } \q .
130 not_quote = not_paren_star_quote_special | letter | digit | '(' | ')' | '*' .
128 not_paren_star_quote_special = '!' | '"' | '#' | '$' | '%' | '&' | '+' | ',' |
                                   '-' | '.' | '/' | ':' | ';' | '<' | '=' | '>' |
                                   '?' | '@' | '[' | '\' | ']' | '^' | '_' | '`' |
                                   '{' | '|' | '}' | '~' .
124 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' .
120 digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' .
137 encoded_string_literal = '"' encoded_character { encoded_character } '"' .
122 encoded_character = octet octet octet octet .
133 octet = hex_digit hex_digit .
123 hex_digit = digit | 'a' | 'b' | 'c' | 'd' | 'e' | 'f' .
```

The implementor of an *EXPRESS* language parser shall specify the maximum number of characters of a simple string literal which can be read by that implementation, using annex E.

The implementor of an *EXPRESS* language parser shall also specify the maximum number of octets (must be a multiple of four) of an encoded string literal which can be read by that implementation, using annex E.

EXAMPLES

11 – Valid simple string literals

```
'Baby needs a new pair of shoes!'
```
Reads . . . Baby needs a new pair of shoes!

```
'Ed''s Computer Store'
```
Reads . . . Ed's Computer Store

12 – Invalid simple string literals

```
'Ed's Computer Store'
```
There must always be an even number of quote marks.

```
'Ed''s Computer
Store'
```
Spans a physical line

13 – Valid encoded string literals

```
"00000041"
```
Reads A

```
"000000C5"
```
Reads Å

14 – Invalid encoded string literals

"000041"
Octets must be supplied in groups of four

"00000041 000000C5"
Only hexadecimal characters are allowed between ""

7.5.5 Logical literal

A logical literal represents a value of a logical or boolean data type and is one of the built-in constants TRUE, FALSE or UNKNOWN.

NOTE – UNKNOWN is not compatible with a boolean data type.

Syntax:

```
242 logical_literal = FALSE | TRUE | UNKNOWN .
```

8 Data types

This clause defines the data types provided as part of the language. Every attribute, local variable or formal parameter has an associated data type.

Data types are classified as simple data types, aggregation data types, named data types, constructed data types and generalized data types. Data types are also classified according to their usage as base data types, parameter data types and underlying data types. The relationships between these two classifications are described in 8.6.

The operations that may be performed on values of these data types are defined in clause 12.

8.1 Simple data types

The simple data types define the domains of the atomic data units in *EXPRESS*. That is, they cannot be further subdivided into elements that *EXPRESS* recognizes. The simple data types are NUMBER, REAL, INTEGER, STRING, BOOLEAN, LOGICAL and BINARY.

8.1.1 Number data type

The NUMBER data type has as its domain all numeric values in the language. The NUMBER data type shall be used when a more specific numeric representation is not important.

Syntax:

```
248 number_type = NUMBER .
```

EXAMPLE 15 – Since we may not know the context of `size` we do not know how to correctly represent it, e.g. the size of the crowd at a football game would be an INTEGER, whereas the area of the pitch would be a REAL.

```
size : NUMBER ;
```

NOTE – In future editions of this standard there may be further specializations of the NUMBER data type, for example complex numbers.

8.1.2 Real data type

The REAL data type has as its domain all rational, irrational and scientific real numbers. It is a specialization of the NUMBER data type.

Syntax:

```
265 real_type = REAL [ '(' precision_spec ')' ] .
255 precision_spec = numeric_expression .
```

Rational and irrational numbers have infinite resolution and are exact. Scientific numbers represent quantities which are known only to a specified precision. The `precision_spec` is stated in terms of significant digits.

A real number literal is represented by a mantissa and optional exponent. The number of digits making up the mantissa when all leading zeros have been removed is the number of significant digits. The known precision of a value is the number of leading digits that are necessary to the application.

Rules and restrictions:

a) The `precision_spec` gives the minimum number of digits of resolution that are required. This expression shall evaluate to a positive integer value.

b) When no resolution specification is given the precision of the real number is unconstrained.

8.1.3 Integer data type

The INTEGER data type has as its domain all integer numbers. It is a specialization of the REAL data type.

Syntax:

```
227 integer_type = INTEGER .
```

EXAMPLE 16 – This example uses an INTEGER data type to represent an attribute named `nodes`. The domain of this attribute is all integers, with no further constraint.

```
ENTITY foo;
  nodes : INTEGER;
```

```
  ...
END_ENTITY;
```

8.1.4 Logical data type

The LOGICAL data type has as its domain the three literals TRUE, FALSE and UNKNOWN.

Syntax:

```
243 logical_type = LOGICAL .
```

The following ordering holds for the values of the LOGICAL data type: FALSE < UNKNOWN < TRUE. The LOGICAL data type is compatible with the BOOLEAN data type, except that the value UNKNOWN cannot be assigned to a boolean variable.

8.1.5 Boolean data type

The BOOLEAN data type has as its domain the two literals TRUE and FALSE. The BOOLEAN data type is a specialization of the LOGICAL data type.

Syntax:

```
173 boolean_type = BOOLEAN .
```

The same ordering holds for values of the BOOLEAN data type as for values of the LOGICAL data type, that is: FALSE < TRUE.

EXAMPLE 17 – In this example, an attribute named `planar` is represented by the BOOLEAN data type. The value for `planar` associated with an instance of `surface` can be either TRUE or FALSE.

```
ENTITY surface;
  planar : BOOLEAN;
  ...
END_ENTITY;
```

8.1.6 String data type

The STRING data type has as its domain sequences of characters. The characters which are permitted to form part of a string value are defined in ISO 10646.

Syntax:

```
293 string_type = STRING [ width_spec ] .
318 width_spec = '(' width ')' [ FIXED ] .
317 width = numeric_expression .
```

A STRING data type may be defined as either fixed or varying width (number of characters). If it is not specifically defined as fixed width (by using the FIXED reserved word in the definition) the string has varying width.

The domain of a fixed width STRING data type is the set of all character sequences of exactly the width specified in the type definition.

The domain of a varying width STRING data type is the set of all character sequences of width less than or equal to the maximum width specified in the type definition.

If no width is specified, the domain is the set of all character sequences, with no constraint on the width of these sequences.

Substrings and individual characters may be addressed using subscripts as described in 12.5.

The case (upper or lower) of letters within a string is significant.

Rules and restrictions:

The `width` expression shall evaluate to a positive integer value.

EXAMPLES

18 – The following defines a varying length string; values of which have no defined maximum length.

```
string1 : STRING;
```

19 – The following defines a string that is a maximum of ten characters in length; values of which may vary in actual length from zero to ten characters.

```
string2 : STRING(10);
```

20 – The following defines a string that is exactly ten characters in length; values of which must contain ten characters.

```
string3 : STRING(10) FIXED;
```

8.1.7 Binary data type

The BINARY data type has as its domain sequences of bits, each bit being represented by 0 or 1.

Syntax:

```
172 binary_type = BINARY [ width_spec ] .
318 width_spec = '(' width ')' [ FIXED ] .
317 width = numeric_expression .
```

A BINARY data type may be defined as either fixed or varying width (number of bits). If it is not specifically defined as fixed width (by using the FIXED reserved word in the definition) the binary data type has varying width.

The domain of a fixed width BINARY data type is the set of all bit sequences of exactly the width specified in the type definition.

The domain of a varying width BINARY data type is the set of all bit sequences of width less than or equal to the maximum width specified in the type definition. If no width is specified, the domain is the set of all bit sequences, with no constraint on the width of these sequences.

Subbinaries and individual bits may be addressed using subscripts as described in 12.3.

Rules and restrictions:

The `width` expression shall evaluate to a positive integer value.

EXAMPLE 21 – The following might be used to hold character font information.

```
ENTITY character;
  representation : ARRAY [1:20] OF BINARY (8) FIXED ;
END_ENTITY;
```

8.2 Aggregation data types

Aggregation data types have as their domains collections of values of a given base data type (see 8.6.1). These base data type values are called elements of the aggregation collection. *EXPRESS* provides for the definition of four kinds of aggregation data types: ARRAY, LIST, BAG and SET. Each kind of aggregation data type attaches different properties to its values.

– An ARRAY is a fixed-size ordered collection. It is indexed by a sequence of integers.

EXAMPLE 22 – A transformation matrix (for geometry) may be defined as an array of arrays (of numbers).

– A LIST is a sequence of elements which can be accessed according to their position. The number of elements in a list may vary, and can be constrained by the definition of the data type.

EXAMPLE 23 – The operations of a process plan might be represented as a list. The operations are ordered, and operations can be added to or removed from a process plan.

– A BAG is an unordered collection in which duplication is allowed. The number of elements in a bag may vary, and can be constrained by the definition of the data type.

EXAMPLE 24 – The collection of fasteners used in an assembly problem could be represented as a bag. There might be a number of elements which are equivalent bolts, but which one is used in a particular hole is unimportant.

– A SET is an unordered collection of elements in which no two elements are instance equal. The number of elements in a set may vary, and can be constrained by the definition of the data type.

EXAMPLE 25 – The population of people in this world is a set.

NOTE – *EXPRESS* aggregations are one dimensional. Objects usually considered to have multiple dimensions (such as mathematical matrices) can be represented by an aggregation data type whose base type is another aggregation data type. Aggregation data types can be thus nested to an arbitrary depth, allowing any number of dimensions to be represented.

EXAMPLE 26 – One could define a `LIST[1:3] OF ARRAY[5:10] OF INTEGER`, which would in effect have two dimensions.

8.2.1 Array data type

An ARRAY data type has as its domain indexed, fixed-size collections of like elements. The lower and upper bounds, which are integer-valued expressions, define the range of index values, and thus the size of each array collection. An ARRAY data type definition may optionally specify that an array value cannot contain duplicate elements. It may also specify that an array value need not contain an element at every index position.

Syntax:

```
165 array_type = ARRAY bound_spec OF [ OPTIONAL ] [ UNIQUE ] base_type .
176 bound_spec = '[' bound_1 ':' bound_2 ']' .
174 bound_1 = numeric_expression .
175 bound_2 = numeric_expression .
171 base_type = aggregation_types | simple_types | named_types .
```

Given that m is the lower bound and n is the upper bound, there are exactly $n - m + 1$ elements in the array. These elements are indexed by subscripts from m to n, inclusive (see 12.6.1).

NOTE 1 – The bounds may be positive, negative or zero, but may not be indeterminate (?) (see 14.2).

Rules and restrictions:

a) Both expressions in the bound specification, `bound_1` and `bound_2`, shall evaluate to integer values. Neither shall evaluate to the indeterminate (?) value.

b) `bound_1` gives the lower bound of the array. This shall be the lowest index which is valid for an array value of this data type.

c) `bound_2` gives the upper bound of the array. This shall be the highest index which is valid for an array value of this data type.

d) `bound_1` shall be less than or equal to `bound_2`.

e) If the OPTIONAL keyword is specified, an array value of this data type may have the indeterminate (?) value at one or more index positions.

f) If the OPTIONAL keyword is not specified, an array value of this data type shall not contain an indeterminate (?) value at any index position.

g) If the UNIQUE keyword is specified, each element in an array value of this data type shall be different from (i.e., not instance equal to) every other element in the same array value.

NOTE 2 – Both OPTIONAL and UNIQUE may be specified in the same ARRAY data type definition. This does not preclude multiple indeterminate (?) values from occurring in a single array value. This is because comparisons between indeterminate (?) values result in UNKNOWN so the uniqueness constraint is not violated.

EXAMPLE 27 – This example shows how a multi-dimensioned array is declared.

```
sectors : ARRAY [ 1 : 10 ] OF       -- first dimension
  ARRAY [ 11 : 14 ] OF              -- second dimension
    UNIQUE something;
```

The first array has 10 elements of data type `ARRAY[11:14] OF UNIQUE something`. There is a total of 40 elements of data type `something` in the attribute named `sectors`. Within each `ARRAY[11:14]`, no duplicates may occur; however, the same `something` instance may occur in two different `ARRAY[11:14]` values within a single value for the attribute named `sectors`.

8.2.2 List data type

A LIST data type has as its domain sequences of like elements. The optional lower and upper bounds, which are integer-valued expressions, define the minimum and maximum number of elements that can be held in the collection defined by a LIST data type. A LIST data type definition may optionally specify that a list value cannot contain duplicate elements.

Syntax:

```
237 list_type = LIST [ bound_spec ] OF [ UNIQUE ] base_type .
176 bound_spec = '[' bound_1 ':' bound_2 ']' .
174 bound_1 = numeric_expression .
175 bound_2 = numeric_expression .
171 base_type = aggregation_types | simple_types | named_types .
```

Rules and restrictions:

a) The `bound_1` expression shall evaluate to an integer value greater than or equal to zero. It gives the lower bound, which is the minimum number of elements that can be in a list value of this data type. `bound_1` shall not produce the indeterminate (`?`) value.

b) The `bound_2` expression shall evaluate to an integer value greater than or equal to `bound_1`, or an indeterminate (`?`) value. It gives the upper bound, which is the maximum number of elements that can be in a list value of this data type.

If this value is indeterminate (`?`) the number of elements in a list value of this data type is not bounded from above.

c) If the `bound_spec` is omitted, the limits are `[0:?]`.

d) If the UNIQUE keyword is specified, each element in a list value of this data type shall be different from (i.e., not instance equal to) every other element in the same list value.

EXAMPLE 28 – This example defines a list of arrays. The list can contain zero to ten arrays. Each array of ten integers shall be different from all other arrays in a particular list.

```
complex_list : LIST[0:10] OF UNIQUE ARRAY[1:10] OF INTEGER;
```

8.2.3 Bag data type

A BAG data type has as its domain unordered collections of like elements. The optional lower and upper bounds, which are integer-valued expressions, define the minimum and maximum number of elements that can be held in the collection defined by a BAG data type.

Syntax:

```
170 bag_type = BAG [ bound_spec ] OF base_type .
176 bound_spec = '[' bound_1 ':' bound_2 ']' .
174 bound_1 = numeric_expression .
175 bound_2 = numeric_expression .
171 base_type = aggregation_types | simple_types | named_types .
```

Rules and restrictions:

a) The `bound_1` expression shall evaluate to an integer value greater than or equal to zero. It gives the lower bound, which is the minimum number of elements that can be in a bag value of this data type. `bound_1` shall not produce the indeterminate (`?`) value.

b) The `bound_2` expression shall evaluate to an integer value greater than or equal to `bound_1`, or an indeterminate (`?`) value. It gives the upper bound, which is the maximum number of elements that can be in a bag value of this data type.

If this value is indeterminate (`?`) the number of elements in a bag value of this data type is not be bounded from above.

c) If the `bound_spec` is omitted, the limits are `[0:?]`.

EXAMPLE 29 – This example defines an attribute as a bag of point (where point is a named data type assumed to have been declared elsewhere).

```
a_bag_of_points : BAG OF point;
```

The value of the attribute named `a_bag_of_points` can contain zero or more points. The same point instance may appear more than once in the value of `a_bag_of_points`.

If the value is required to contain at least one element, the specification can provide a lower bound, as in:

```
a_bag_of_points : BAG [1:?] OF point;
```

The value of the attribute named `a_bag_of_points` now must contain at least one point.

8.2.4 Set data type

A SET data type has as its domain unordered collections of like elements. The SET data type is a specialization of the BAG data type. The optional lower and upper bounds, which are integer-valued expressions, define the minimum and maximum number of elements that can be held in

the collection defined by a SET data type. The collection defined by SET data type shall not contain two or more elements which are instance equal.

Syntax:

```
285 set_type = SET [ bound_spec ] OF base_type .
176 bound_spec = '[' bound_1 ':' bound_2 ']' .
174 bound_1 = numeric_expression .
175 bound_2 = numeric_expression .
171 base_type = aggregation_types | simple_types | named_types .
```

Rules and restrictions:

a) The `bound_1` expression shall evaluate to an integer value greater than or equal to zero. It gives the lower bound, which is the minimum number of elements that can be in a set value of this data type. `bound_1` shall not produce the indeterminate (`?`) value.

b) The `bound_2` expression shall evaluate to an integer value greater than or equal to `bound_1`, or an indeterminate (`?`) value. It gives the upper bound, which is the maximum number of elements that can be in a set value of this data type.

If this value is indeterminate (`?`) the number of elements in a set value of this data type is not be bounded from above.

c) If the `bound_spec` is omitted, the limits are `[0:?]`.

d) Each element in an occurrence of a SET data type shall be different from (i.e., not instance equal to) every other element in the same set value.

EXAMPLE 30 – This example defines an attribute as a set of points (a named data type assumed to have been declared elsewhere).

```
a_set_of_points : SET OF point;
```

The attribute named `a_set_of_points` can contain zero or more points. Each point instance (in the set value) is required to be different from every other point in the set.

If the value is required to have no more than 15 points, the specification can provide an upper bound, as in:

```
a_set_of_points : SET [0:15] OF point;
```

The value of the attribute named `a_set_of_points` now may contain no more than 15 points.

8.2.5 Value uniqueness on aggregates

Uniqueness among the elements of an aggregation is based upon instance comparison (see 12.2.2). Aggregates can be constrained to be value unique among their elements through the use of the VALUE_UNIQUE function (see 15.29).

EXAMPLE 31 – A set is constrained to be value unique.

```
TYPE value_unique_set = SET OF a;
WHERE
   wr1 : value_unique(SELF);
END_TYPE;
```

NOTE – Modeller-defined value uniqueness can be specified via a pair of functions, called for example `my_equal` and `my_unique`, as shown in the following pseudo-code.

```
FUNCTION my_equal(v1,v2: GENERIC:gen): LOGICAL;
 (* Returns TRUE if v1 'equals' v2 *)
END_FUNCTION;

FUNCTION my_unique(c: AGGREGATE OF GENERIC): LOGICAL;
  (* Returns FALSE if two elements of c have the same 'value'
     Else returns UNKNOWN if any element comparison is UNKNOWN
     Otherwise returns TRUE *)
LOCAL
  result   : LOGICAL;
  unknownp : BOOLEAN := FALSE;
END_LOCAL;
IF (SIZEOF(c) = 0) THEN
  RETURN(TRUE); END_IF;
REPEAT i := LOINDEX(c) TO (HIINDEX(c) - 1);
  REPEAT j := (i+1) TO HIINDEX(c);
    result := my_equal(c[i], c[j]);
    IF (result = TRUE) THEN
      RETURN(FALSE); END_IF;
    IF (result = UNKNOWN) THEN
      unknownp := TRUE; END_IF;
  END_REPEAT;
END_REPEAT;
IF unknownp THEN
  RETURN(UNKNOWN);
ELSE
  RETURN(TRUE);
END_IF;
END_FUNCTION;
```

The function `my_equal` should have the following properties which enable the building of equivalence classes. In the following $\mathcal{S}$ is the set of objects under consideration and `my_equal(i, j)`, where `i` and `j` are in $\mathcal{S}$, returns one of [FALSE, UNKNOWN, TRUE].

a) `my_equal(i, i)` is TRUE for all `i` in $\mathcal{S}$ (since indeterminate (`?`) is not in $\mathcal{S}$, this does not require `my_equal(?, ?)` to be TRUE);

b) `my_equal(i, j)` = `my_equal(j, i)` for all `i, j` in $\mathcal{S}$;

c) (`my_equal(i, j)` = TRUE) AND (`my_equal(j, k)` = TRUE) implies (`my_equal(i, k)` = TRUE) for all `i, j, k` in $\mathcal{S}$.

8.3 Named data types

The named data types are the data types that may be declared in a formal specification. There are two kinds of named data types: entity data types and defined data types. This clause covers the referencing of named data types; the declaration of these data types is covered in clause 9.

8.3.1 Entity data type

Entity data types are established by ENTITY declarations (see 9.2). An entity data type is assigned an entity identifier by the user. An entity data type is referenced by this identifier.

Syntax:

```
147 entity_ref = entity_id .
```

Rules and restrictions:

`entity_ref` shall be a reference to an entity which is visible in the current scope (see clause 10).

EXAMPLE 32 – This example uses a `point` entity data type as the representation of an attribute.

```
ENTITY point;
    x, y, z : REAL;
END_ENTITY;

ENTITY line;
    p0, p1 : point;
END_ENTITY;
```

The line entity has two attributes named `p0` and `p1`. The data type of each of these attributes is `point`.

8.3.2 Defined data type

Defined data types are declared by TYPE declarations (see 9.1). A defined data type is assigned a type identifier by the user. A defined data type is referenced by this identifier.

Syntax:

```
154 type_ref = type_id .
```

Rules and restrictions:

`type_ref` shall be the name of a defined data type which is visible in the current scope (see clause 10).

EXAMPLE 33 – The following is a defined data type used to indicate the units of measure associated with an attribute.

```
TYPE volume = REAL;
END_TYPE;
```

```
ENTITY PART;
...
  bulk : volume;
END_ENTITY;
```

The attribute named `bulk` is represented as a real number, but the use of the defined data type, `volume`, helps to clarify the meaning and context of the real number; i.e., it means volume, rather than some other thing which might be represented by a REAL.

8.4 Constructed data types

There are two kinds of constructed data types in *EXPRESS*: ENUMERATION data types and SELECT data types. These data types have similar syntactic structures and are used to provide underlying representations of defined data types (see 9.1).

NOTE – The *EXPRESS* constructed data types can only be used as representations for defined data types.

8.4.1 Enumeration data type

An ENUMERATION data type has as its domain an ordered set of names. The names represent values of the enumeration data type. These names are designated by `enumeration_id`s and are referred to as enumeration items.

Syntax:

```
201 enumeration_type = ENUMERATION OF '(' enumeration_id { ',' enumeration_id } ')' .
```

Two different ENUMERATION data types may contain the same `enumeration_id`. In this case, any reference to the `enumeration_id` (e.g. in an expression) shall be pre-qualified with the data type identifier to ensure that the reference is unambiguous. The reference then appears as: `type_id.enumeration_id`.

NOTE – There is always a `type_id` available, because *EXPRESS* allows an ENUMERATION data type only as the underlying representation of a defined data type.

Rules and restrictions:

a) For comparison purposes, the ordering of the values of an enumeration type shall be determined by their relative position in the `enumeration_id` list: the first occurring item shall be less than the second; the second less than the third, etc.

b) Comparison between values of different ENUMERATION data types is undefined.

c) An enumeration type shall only be used as the underlying type of a defined data type.

EXAMPLE 34 – This example uses ENUMERATION data types to show how different kinds of vehicles might travel.

```
TYPE car_can_move = ENUMERATION OF
```

```
     (left, right, backward, forward);
END_TYPE;

TYPE plane_can_move = ENUMERATION OF
     (left, right, backward, forward, up, down);
END_TYPE;
```

The enumeration item `left` has two independent definitions, one being given by each type of which it is a component. There is no connection between these two definitions of the identifier `left`. A reference to `left` or `right`, by itself, is ambiguous. To resolve the ambiguity, a reference to either of these values must be qualified by the type name, e.g., `car_can_move.left`.

8.4.2 Select data type

A SELECT data type has as its domain the union of the domains of the named data types in its select list. The SELECT data type is a generalization of each of the named data types in its select list.

Syntax:

```
284 select_type = SELECT '(' named_types { ',' named_types } ')' .
```

Rules and restrictions:

a) Each item in the select list shall be an entity data type or a defined data type.

b) A SELECT data type shall only be used as the underlying type of a defined data type.

NOTE – The value of a SELECT data type may be a value of more than one of the named data types specified in the select list for that select data type. This situation only occurs when the relevant named data types are part of a common inheritance graph (9.2.3).

EXAMPLE 35 – If *a* and *b* are subtypes of *c* and they are related by an ANDOR expression, and we have a type defined by `SELECT (a,b);`, we may have the value of the SELECT data type which is an *a* and a *b* at the same time.

EXAMPLE 36 – A choice must be made among several types of things in a given context.

```
TYPE
    attachment_method = SELECT(permanent_attachment, temporary_attachment);
END_TYPE;

TYPE permanent_attachment = SELECT(glue, weld);
END_TYPE;

TYPE temporary_attachment = SELECT(nail, screw);
END_TYPE;

ENTITY nail;
  body_length : REAL;
  head_area   : REAL;
```

```
END_ENTITY;

ENTITY screw;
  body_length : REAL;
  pitch       : REAL;
END_ENTITY;

ENTITY glue;
  composition : material_composition;
  solvent     : material_composition;
END_ENTITY;

ENTITY weld;
  composition : material_composition;
END_ENTITY;

ENTITY wall_mounting;
  mounting  : product;
  on        : wall;
  using     : attachment_method;
END_ENTITY;
```

A `wall_mounting` attaches a `product` onto a `wall` using either a permanent or temporary attachment method, both of which are further subdivided. A value of a `wall_mounting` will have an `using` attribute which is a value of one of `nail`, `screw`, `glue` or `weld`.

8.5 Generalized data types

Syntax:

```
211 generalized_types = aggregate_type | general_aggregation_types | generic_type .
```

The generalized data types are used to specify a generalization of certain other data types, and can only be used in certain very specific contexts. The GENERIC type is a generalization of all data types. The AGGREGATE data type is a generalization of all aggregation data types. The general aggregate data types are generalizations of aggregation data types which relax some of the constraints normally applied to aggregation data types. All of these data types are defined in 9.5.3.

8.6 Data type usage classification

Data types are used in three different ways in *EXPRESS*: Base data types are used as representations for attributes and aggregation elements; parameter data types are used as representations for formal parameters to functions and procedures; and underlying data types are used as representations for defined data types. Some classes of data types can be used in any of these ways, while others can only be used in certain contexts. These distinctions are summarized in table 7.

Table 7 – The use of data types

Type	a	b	c
Simple Data Types	•	•	•
Aggregation Data Types	•	•	•
Named Data Types	•	•	•
Constructed Data Types			•
Generalized Data Types		•	

a) Base data types - representation of an attribute or of aggregation elements.

b) Parameter data types - representation of a formal parameter, local variable or function result.

c) Underlying data types - representation of a defined type (see 9.1).

8.6.1 Base data types

Base data types are used as the representation of attributes or as the base types of aggregation data types.

The base data types are the simple data types, the aggregation data types and the named data types.

Syntax:

```
171 base_type = aggregation_types | simple_types | named_types .
```

8.6.2 Parameter data types

Parameter data types are used as the representation of formal parameters to algorithms (functions and procedures). The parameter data types may also be used to represent the return types of functions and the local variables declared in algorithms.

The parameter data types are the simple data types, the named data types and the generalized data types.

Syntax:

```
253 parameter_type = generalized_types | named_types | simple_types .
```

8.6.3 Underlying data types

Underlying data types are used as the representation for defined data types.

The underlying data types are the simple data types, the aggregation data types, the constructed data types and the defined data types.

Syntax:

```
309 underlying_type = constructed_types | aggregation_types | simple_types |
                      type_ref .
```

9 Declarations

This clause defines the various declarations available in *EXPRESS*. An *EXPRESS* declaration creates a new *EXPRESS* item and associates an identifier with it. The *EXPRESS* item may be referenced elsewhere by writing the name associated with it (see clause 10).

The principle capabilities of *EXPRESS* are found in the following declarations:

- Type;
- Entity;
- Schema;
- Constant;
- Function;
- Procedure;
- Rule.

Declarations may be either explicit or implicit. This clause describes explicit declarations. Implicit declarations are described in this and subsequent clauses, along with the items and conditions under which they are established.

9.1 Type declaration

A type declaration creates a defined data type (see 8.3.2) and declares an identifier to refer to it. Specifically, the `type_id` is declared as the name of a defined data type. The representation of this data type is the `underlying_type`. The domain of the defined data type is the same as the domain of the `underlying_type`, further constrained by the `where_clause` (if present). The defined data type is a specialization of the underlying type and is therefore compatible with the underlying type.

NOTE 1 – Multiple defined data types may be associated with the same representation. The names can help the reader to understand the intent (or context) of the use of the `underlying_type`.

Syntax:

```
304 type_decl = TYPE type_id '=' underlying_type ';' [ where_clause ] END_TYPE ';' .
309 underlying_type = constructed_types | aggregation_types | simple_types |
                      type_ref .
```

EXAMPLE 37 – The following declaration declares a defined data type named `person_name` with an underlying representation of STRING. The defined type `person_name` is then available for use as the representation of attributes, local variables and formal parameters. This conveys more meaning than simply using STRING.

```
TYPE person_name = STRING;
END_TYPE;
```

Domain rules (WHERE clause)

Domain rules specify constraints that restrict the domain of the defined data type. The domain of the defined data type is the domain of its underlying representation constrained by the domain rule(s). Domain rules follow the WHERE keyword.

Syntax:

```
315 where_clause = WHERE domain_rule ';' { domain_rule ';' } .
```

Each `domain_rule` may be given a rule label. The referencing of rule labels is outside the scope of this part of ISO 10303.

NOTE 2 – When given, rule labels may be used to identify rules to implementations, for example, in documentation, error reports and enforcement specifications. The labelling of rules for this purpose is encouraged.

Rules and restrictions:

a) Each domain rule shall evaluate to either a logical (TRUE, FALSE or UNKNOWN) value or indeterminate (?).

b) The keyword SELF (see 14.5) shall appear at least once in each domain rule. A domain rule is evaluated for a particular value in the domain of the underlying type by replacing each occurrence of SELF in the rule with that value.

c) A domain rule shall be asserted when the expression evaluates to a value of TRUE; it shall be violated when the expression evaluates to a value of FALSE; and it shall be neither violated nor asserted when the expression evaluates to a indeterminate (?) or UNKNOWN value.

d) The domain of the defined data type consists of all values of the domain of the underlying type which do not violate any of the domain rules.

e) Domain rule labels shall be unique within a given TYPE declaration.

EXAMPLE 38 – A defined data type could be created which constrains the underlying integer data type to allow only positive integers.

```
TYPE positive = INTEGER;
WHERE
  notnegative : SELF > 0;
END_TYPE;
```

Any attribute, local variable or formal parameter declared to be of type `positive` is then constrained to have only positive integer values.

9.2 Entity declaration

An ENTITY declaration creates an entity data type and declares an identifier to refer to it.

Each attribute represents a property of an entity and may be associated with a value in each entity instance. The data type of the attribute establishes the domain of possible values.

Each constraint represents one of the following properties of the entity:

– limits on the number, kind and organization of values of attributes. These are specified in the attribute declarations.

– required relationships among attribute values or restrictions on the allowed attribute values for a given instance. These appear in the where clause, and are referred to as domain rules.

– required relationships among attribute values over all instances of the entity data type. These appear in:

- the uniqueness clause, where they are referred to as uniqueness constraints,
- the inverse clause, where they are referred to as cardinality constraints,
- global rules (see 9.6).

– required relationships among instances of more than one entity type. These do not appear in the entity declaration itself, but rather as global rules (see 9.6).

Syntax:

```
196 entity_decl = entity_head entity_body END_ENTITY ';' .
197 entity_head = ENTITY entity_id [ subsuper ] ';' .
194 entity_body = { explicit_attr } [ derive_clause ] [ inverse_clause ]
                  [ unique_clause ] [ where_clause ] .
```

Rules and restrictions:

a) Each attribute identifier and label declared in the entity declaration shall be unique within the declaration.

b) A subtype shall not declare an attribute having the same identifier as an attribute of one of its supertypes, except when a subtype redeclares an attribute inherited from of one of its supertypes (see 9.2.3.4).

9.2.1 Attributes

The attributes of an entity data type represent an entity's essential traits, qualities or properties. An attribute declaration establishes a relationship between the entity data type and the data type referenced by the attribute.

The name of an attribute represents the role played by its associated value in the context of the entity in which it appears.

There are three kinds of attribute:

Explicit: An attribute whose value shall be supplied by an implementation in order to create an entity instance.

Derived: An attribute whose value is computed in some manner.

Inverse: An attribute whose value consists of the entity instances which use the entity in a particular role.

Every attribute establishes relationships between an instance of the declaring entity data type and some other instance or instances. An attribute represented by a non-aggregation data type establishes a simple relationship to this data type. An attribute represented by an aggregation data type establishes both collective relationships to aggregate values and distributive relationships to the elements of those aggregate values. In addition, every attribute establishes an implicit inverse relationship between its base data type and the declaring entity data type.

NOTE – See annex G for further discussion of these relationships.

9.2.1.1 Explicit attribute

An explicit attribute represents a property whose value shall be supplied by an implementation in order to create an instance. Each explicit attribute identifies a distinct property. An explicit attribute declaration creates one or more explicit attributes having the indicated domain, and assigns an identifier to each.

Syntax:

```
203 explicit_attr = attribute_decl { ',' attribute_decl } ':' [ OPTIONAL ]
                    base_type ';' .
167 attribute_decl = attribute_id | qualified_attribute .
171 base_type = aggregation_types | simple_types | named_types .
```

NOTE 1 – The syntax for `qualified_attribute` provides for attribute redeclaration, which is described in 9.2.3.4.

Rules and restrictions:

a) Unless an explicit attribute is declared OPTIONAL, every instance of the entity data type shall have a value for that attribute.

b) The keyword OPTIONAL indicates that, in a given entity instance, the attribute need not have a value. If the attribute has no value, the value is said to be indeterminate (`?`).

OPTIONAL indicates that the attribute is always meaningful for instances of this entity type, but that for some instances there may be no value which plays the role specified by the attribute. OPTIONAL does not indicate that the attribute is meaningless for some instances of the entity data type.

NOTES

2 – The case where an attribute is meaningless for some instances is properly modelled by subtyping (see 9.2.3).

3 – Attention must be given to references to optional attributes, particularly in rules, since they may have no value. The EXISTS built-in function can be used to determine whether a value is present or the NVL built-in function can be used to provide a default value for computation. If neither is used, unexpected results may be obtained.

EXAMPLE 39 – The following declarations are equivalent:

```
ENTITY point;
  x, y, z : REAL;
END_ENTITY;

ENTITY point;
  x : REAL;
  y : REAL;
  z : REAL;
END_ENTITY;
```

9.2.1.2 Derived attribute

A derived attribute represents a property whose value is computed by evaluating an expression. Derived attributes are declared following the DERIVE keyword. The declaration consists of the attribute identifier, its representation type and an expression to be used to compute the attribute value.

Syntax:

```
190 derived_attr = attribute_decl ':' base_type ':=' expression ';' .
167 attribute_decl = attribute_id | qualified_attribute .
171 base_type = aggregation_types | simple_types | named_types .
```

NOTE – The syntax for `qualified_attribute` provides for attribute redeclaration, which is described in 9.2.3.4.

The expression may refer to any attribute, constant (including SELF) or function identifier which is in scope.

Rules and restrictions:

a) The `expression` shall be assignment compatible (see 13.3) with the data type of the attribute.

b) For a particular entity instance, the value of the derived attribute is determined by evaluating the expression, replacing each occurrence of SELF with the current instance, and each occurrence of an attribute reference with the corresponding attribute value.

EXAMPLE 40 – In the following example, a circle is defined by a centre, axis and radius. In addition to these explicit attributes, there is a need to account for important properties such as the

area and perimeter. This is accomplished by defining them as derived attributes, giving the values as expressions.

```
ENTITY circle;
  centre : point;
  radius : REAL;
  axis   : vector;
DERIVE
  area      : REAL := PI*radius**2;
  perimeter : REAL := 2.0*PI*radius;
END_ENTITY;
```

9.2.1.3 Inverse attribute

If another entity has established a relationship with the current entity by way of an explicit attribute, an inverse attribute may be used to describe that relationship in the context of the current entity. This inverse attribute may also be used to constrain the relationship further.

Inverse attributes are declared following the INVERSE keyword. Each inverse attribute shall be specified individually.

Cardinality constraints on the inverse relationship are specified by the bound specification for the inverse attribute in the same way as they are for explicit attributes.

NOTE 1 – Annex G provides further information on the relationship between explicit attributes and inverse attributes.

An inverse attribute is represented by either an entity data type or a BAG or SET whose base type is an entity data type. The entity data type is referred to as the referencing entity.

An inverse attribute declaration also names an explicit attribute of the referencing entity. For a particular instance of the current entity data type, the value of the inverse attribute consists of the instance or instances of the referencing entity data types which use the current instance in the role specified.

Each of the three possible representation types for an inverse imposes certain constraints on the relationship between the two entities.

Bag data type: The bound specification, if present, specifies the minimum and maximum number of instances of the referencing entity which may use an instance of the current entity. Since a bag may contain a single instance more than once, one or more instances may reference the current instance, and a particular instance may reference the current instance more than once.

NOTES

2 – If the inverted attribute is represented by a non-unique aggregation data type, i.e., a list or array which does not specify the UNIQUE keyword or a bag, a particular instance of the current entity may be used more than once by a particular instance of the referencing entity.

3 – If the inverted attribute is represented by a unique aggregation data type, i.e., a list or array which does specify the UNIQUE keyword or a set, a particular instance of the current entity may

only be used once by a particular instance of the referencing entity.

Optionality of the inverse attribute is expressed by specifying a lower bound of 0, indicating that a given instance of the current entity need not be referenced by any instance of the referencing entity.

Set data type: As for BAG, with the additional constraint that the referencing instances must be instance unique. This restriction also means that a particular referencing instance can only use the current instance once in the inverted role.

NOTE 4 – If the inverted attribute is represented by a unique aggregation data type, i.e., a list or array which specifies the UNIQUE keyword or a set, the inverse adds no further constraint with respect to uniqueness.

Entity data type: The inverse attribute contains exactly that one instance of the referencing entity data type which uses the current instance in the specified role. The cardinality of the inverse relationship is 1 : 1 in this case.

Syntax:

```
234 inverse_attr = attribute_decl ':' [ ( SET | BAG ) [ bound_spec ] OF ] entity_ref
                   FOR attribute_ref ';' .
167 attribute_decl = attribute_id | qualified_attribute .
176 bound_spec = '[' bound_1 ':' bound_2 ']' .
174 bound_1 = numeric_expression .
175 bound_2 = numeric_expression .
```

Rules and restrictions:

a) The entity which defines the direct relationship to the current entity being declared shall do so as an explicit attribute.

b) The data type of the explicit attribute in the entity defining the direct relationship shall be the current entity being declared, one of its supertypes, or an aggregation data type using the current entity or one of its supertypes as its base type.

EXAMPLE 41 – Assuming we have the following declaration for a door assembly:

```
ENTITY door;
  handle : knob;
  hinges : SET [1:?] OF hinge;
END_ENTITY;
```

then we may wish to constrain the declaration of knob such that knobs can only exist if they are used in the role of `handle` in one instance of a door.

```
ENTITY knob;
...
INVERSE
  opens : door FOR handle;
```

```
END_ENTITY;
```

On the other hand, we may merely wish to specify that a knob is used by zero or one doors (e.g. it's either on a door or has yet to be attached to a door).

```
ENTITY knob;
...
INVERSE
  opens : SET [0:1] OF door FOR handle;
END_ENTITY;
```

9.2.2 Local rules

Local rules are assertions on the domain of entity instances and thus apply to all instances of that entity data type. There are two kinds of local rules. Uniqueness rules control the uniqueness of attribute values among all instances of a given entity data type. Domain rules describe other constraints on or among the values of the attributes of each instance of a given entity data type.

Each of the local rules may be given a rule label. The referencing of rule labels is outside the scope of this part of ISO 10303.

NOTE – When given, rule labels may be used to identify rules to implementations, for example, in documentation, error reports and enforcement specifications. The labelling of rules for this purpose is encouraged.

9.2.2.1 Uniqueness rule

A uniqueness constraint for individual attributes or combinations of attributes may be specified in a uniqueness rule. The uniqueness rules follow the UNIQUE keyword, and specify either a single attribute name or a list of attribute names. A rule which specifies a single attribute name, called a simple uniqueness rule, specifies that no two instances of the entity data type in the domain shall use the same instance for the named attribute. A rule which specifies two or more attribute names, called a joint uniqueness rule, specifies that no two instances of the entity data type shall have the same combination of instances for the named attributes.

NOTE – Comparisons are made via instance equality and not via value equality (see 12.2.2).

Syntax:

```
310 unique_clause = UNIQUE unique_rule ';' { unique_rule ';' } .
311 unique_rule = [ label ':' ] referenced_attribute { ',' referenced_attribute } .
266 referenced_attribute = attribute_ref | qualified_attribute .
```

Rules and restrictions:

When an explicit attribute which is marked as OPTIONAL (see 9.2.1.1) appears in a uniqueness rule, if the attribute has no value for a particular entity instance, the uniqueness rule is neither violated nor asserted, and therefore the entity instance is a member of the domain.

EXAMPLES

42 – If an entity had three attributes called a, b and c, we could have:

```
ENTITY e;
a,b,c : INTEGER ;
UNIQUE
ur1 : a;
ur2 : b;
ur3 : c;
END_ENTITY;
```

This means that two instances of the entity data type being declared cannot have the same value for a, for b or for c.

43 – A person_name entity might look like:

```
ENTITY person_name;
  last     : STRING;
  first    : STRING;
  middle   : STRING;
  nickname : STRING;
END_ENTITY;
```

and it might be used as:

```
ENTITY employee;
 badge : NUMBER;
 name  : person_name;
  ...
UNIQUE
 ur1: badge, name;
  ...
END_ENTITY;
```

In this example, two instances of the person_name entity could have the same set of values for the four attributes. In the case of an employee, however, there is a requirement that badge and name together have to be unique. Thus, two instances of employee may have the same value of badge and the same value of name. However, no two instances of employee may have the same instance of name and the same instance of badge since taken together this combination of instances shall be unique (see 9.6 for a method of describing attribute value uniqueness).

9.2.2.2 Domain rules (WHERE clause)

Domain rules constrain the values of individual attributes or combinations of attributes for every entity instance. All domain rules follow the WHERE keyword.

Syntax:

```
315 where_clause = WHERE domain_rule ';' { domain_rule ';' } .
```

Rules and restrictions:

a) Each domain rule expression shall evaluate to either a logical (TRUE, FALSE, or UNKNOWN) value or indeterminate (?).

b) Every domain rule expression shall include a reference to SELF or attributes declared within the entity or any of its supertypes.

c) An occurrence of the keyword SELF shall refer to an instance of the entity being declared.

d) The domain rule shall be asserted when the expression evaluates to a value of TRUE; it shall be violated when the expression evaluates to a value of FALSE; and it shall be neither violated nor asserted when the expression evaluates to a indeterminate (?) or UNKNOWN value.

e) No domain rule shall be violated for a valid instance of the entity (in the domain).

EXAMPLE 44 – A `unit_vector` requires that the length of the vector be exactly one. This constraint can be specified by:

```
ENTITY unit_vector;
  a, b, c : REAL;
WHERE
  length_1 : a**2 + b**2 + c**2 = 1.0;
END_ENTITY;
```

Optional attributes in domain rules

A domain rule which contains an optional attribute shall be treated this way:

Rules and restrictions:

a) When the attribute has a value, the domain rule shall be evaluated as any other domain rule.

b) When the attribute has no value, the indeterminate (?) value is used as the attribute value in evaluating the domain rule expression. The evaluation of expressions containing the indeterminate (?) value is addressed in clause 12.

EXAMPLE 45 – Consider this variation of example 44.

```
ENTITY unit_vector;
  a, b : REAL;
  c    : OPTIONAL REAL;
WHERE
  length_1 : a**2 + b**2 + c**2 = 1.0;
END_ENTITY;
```

The intent of the domain rule is to ensure that a `unit_vector` is unitized. However, when `c` has the indeterminate (?) value the domain rule always evaluates to UNKNOWN no matter what the values of `a` and `b` are.

The NVL standard function (see 15.18) may be used to supply a reasonable value when the optional attribute is indeterminate (?). When the optional attribute has a value, the NVL function returns that value; otherwise it returns a substitute value.

```
ENTITY unit_vector;
  a, b : REAL;
  c    : OPTIONAL REAL;
WHERE
  length_1 : a**2 + b**2 + NVL(c, 0.0)**2 = 1.0;
END_ENTITY;
```

9.2.3 Subtypes and supertypes

EXPRESS allows for the specification of entities as subtypes of other entities, where a subtype entity is a specialization of its supertype. This establishes an inheritance (i.e., subtype/supertype) relationship between the entities in which the subtype inherits the properties (i.e., attributes and constraints) of its supertype. Successive subtype/supertype relationships establish an inheritance graph in which every instance of a subtype is an instance of its supertype(s).

The inheritance graph established by subtype/supertype relationships shall be acyclic.

An entity declaration which completely defines all the significant properties of that entity declares a simple entity data type. An entity declaration which establishes inheritance relationships with supertypes declares a complex entity data type. A complex entity data type within an inheritance graph shares characteristics of its supertype(s). A complex entity data type may have additional characteristics not contained within its supertype(s).

Syntax:

```
294 subsuper = [ supertype_constraint ] [ subtype_declaration ] .
297 supertype_constraint = abstract_supertype_declaration | supertype_rule .
156 abstract_supertype_declaration = ABSTRACT SUPERTYPE [ subtype_constraint ] .
295 subtype_constraint = OF '(' supertype_expression ')' .
298 supertype_expression = supertype_factor { ANDOR supertype_factor } .
299 supertype_factor = supertype_term { AND supertype_term } .
301 supertype_term = entity_ref | one_of | '(' supertype_expression ')' .
250 one_of = ONEOF '(' supertype_expression { ',' supertype_expression } ')' .
300 supertype_rule = SUPERTYPE subtype_constraint .
```

The following facts pertain to subtype/supertype relationships. These facts refer to a subtype/supertype graph. A subtype/supertype graph is a multiply rooted directed acyclic graph where the nodes represent the entity types and the links represent the subtype/supertype relationships. Following the `SUBTYPE OF` links leads to supertypes while following the `SUPERTYPE OF` links leads to subtypes.

Rules and restrictions:

a) The supertype clause, if present, shall precede the subtype clause, if present.

b) A subtype may have more than one supertype.

c) A supertype may have more than one subtype.

d) A supertype may itself be a subtype of one or more other entity types. That is, paths in the subtype/supertype graph can traverse several nodes.

e) The subtype/supertype relationship shall be transitive. That is, if A is a subtype of B, and B is a subtype of C, A is a subtype of C. The entities which are supertypes of a particular entity type shall be those entities to which it is possible to traverse, starting at the entity type and following the SUBTYPE OF links.

f) A subtype shall not be the supertype of any type in the list of all its supertypes, i.e., the subtype/supertype graph is acyclic.

9.2.3.1 Specifying subtypes

An entity is a subtype if it contains a SUBTYPE declaration. The subtype declaration shall identify all the entity's immediate supertype(s).

Syntax:

```
296 subtype_declaration = SUBTYPE OF '(' entity_ref { ',' entity_ref } ')' .
```

9.2.3.2 Specifying supertypes

An entity is a supertype through either an explicit or implicit specification. An entity is explicitly specified to be a supertype if it contains an ABSTRACT SUPERTYPE declaration and is implicitly specified to be a supertype if it is named in a subtype declaration of at least one other entity.

Syntax:

```
297 supertype_constraint = abstract_supertype_declaration | supertype_rule .
156 abstract_supertype_declaration = ABSTRACT SUPERTYPE [ subtype_constraint ] .
295 subtype_constraint = OF '(' supertype_expression ')' .
298 supertype_expression = supertype_factor { ANDOR supertype_factor } .
299 supertype_factor = supertype_term { AND supertype_term } .
301 supertype_term = entity_ref | one_of | '(' supertype_expression ')' .
250 one_of = ONEOF '(' supertype_expression { ',' supertype_expression } ')' .
300 supertype_rule = SUPERTYPE subtype_constraint .
```

Rules and restrictions:

All subtypes referred to in a supertype expression shall contain a subtype declaration which identifies the current entity as a supertype.

EXAMPLE 46 – The odd numbers are a subtype of the integer numbers, hence the integer numbers are a supertype of the odd numbers.

```
ENTITY integer_number;
  val : INTEGER;
```

```
END_ENTITY;

ENTITY odd_number
  SUBTYPE OF (integer_number);
WHERE
  not_even : ODD(val);
END_ENTITY;
```

9.2.3.3 Attribute inheritance

The attribute identifiers in a supertype are visible within the scope of the subtype (see clause 10). Thus, a subtype inherits all of the attributes of its supertype. This allows the subtypes to specify either constraints or their own attributes using the inherited attribute. If a subtype has more than one supertype, the subtype inherits all of the attributes from all of its supertypes. This is called multiple inheritance.

Rules and restrictions:

An entity shall not declare an attribute with the same name as an attribute inherited from one of its supertypes unless it is redeclaring the inherited attribute (see 9.2.3.4).

When a subtype inherits attributes from two supertypes which are themselves disjoint, it is possible that they have distinct attributes which have the same attribute identifier. The naming ambiguity shall be resolved by prefixing the identifier with the name of the supertype entity from which each attribute is inherited.

EXAMPLE 47 – This example shows how the entity `e12` inherits two attributes named `attr`, and that to identify which of the two attributes is being constrained the attribute name is prefixed.

```
ENTITY e1;
  attr : REAL;
  ...
END_ENTITY;

ENTITY e2;
  attr : BINARY;
  ...
END_ENTITY;

ENTITY e12
SUBTYPE OF (e1,e2);
  ...
WHERE
  positive : SELF\e1.attr > 0.0 ;
             -- attr as declared in e1
END_ENTITY;
```

A subtype may inherit the same attribute from different supertypes which in turn have inherited it from a single supertype. This is called repeated inheritance. In this case the subtype only inherits the attribute once; i.e., there is only one value for this attribute in an instance of this entity data type.

9.2.3.4 Attribute redeclaration

An attribute declared in a supertype can be redeclared in a subtype. The attribute remains in the supertype but the allowed domain of values for that attribute is governed by the redeclaration given in the subtype. The original declaration may be changed in three general ways:

– the data type of the attribute may be changed to a specialization of the original data type (see 9.2.6);

EXAMPLE 48 – A NUMBER data type attribute may be changed to an INTEGER data type or to a REAL data type.

– an optional attribute in the supertype may be changed to a mandatory attribute in the subtype;

– an explicit attribute in the supertype may be changed to a derived attribute in the subtype;

Syntax:

```
262 qualified_attribute = SELF group_qualifier attribute_qualifier .
219 group_qualifier = '\' entity_ref .
169 attribute_qualifier = '.' attribute_ref .
```

Rules and restrictions:

a) The attribute redeclared in the subtype shall be a specialization of the attribute of the same name in the supertype.

b) The name of the redeclared attribute shall be given using syntax of `qualified_attribute`.

c) If an attribute of a supertype is redeclared in two non-mutually exclusive subtypes, an instance which contains both subtypes shall have a single value for that attribute which is valid for both redeclarations. An implementation of an *EXPRESS* language parser which claims level 4 conformance shall test for conflicting redeclarations in subtypes which may coexist in a single instance.

EXAMPLES

49 – Some geometry systems use floating point coordinates while others work in an integer coordinate space.

```
ENTITY point;
  x : NUMBER;
  y : NUMBER;
END_ENTITY;

ENTITY integer_point
  SUBTYPE OF (point);
```

```
  SELF\point.x : INTEGER;
  SELF\point.y : INTEGER;
END_ENTITY;
```

50 – This example shows changing of the elements in an aggregation data type to be unique, reduction of the number of elements in an aggregation data type, and changing an optional attribute to a mandatory one.

```
ENTITY super;
  things : LIST [3:?] OF thing;
  items  : BAG [0:?] OF widget;
  may_be : OPTIONAL stuff;
END_ENTITY;

ENTITY sub
  SUBTYPE OF (super);
  SELF\super.things : LIST [3:?] OF UNIQUE thing;
  SELF\super.items  : SET [1:10] OF widget;
  SELF\super.may_be : stuff;
END_ENTITY;
```

51 – In the following example, a circle is defined by a centre, axis and radius. A variant of circle is defined by the centre and two points through which it passes. These three points represent the data by which this type of circle is defined. In addition to these data there is a need to account for the other important traits – the radius and the axis. This is accomplished by redeclaring them as derived attributes, giving the value as an expression.

```
FUNCTION distance(p1, p2 : point) : REAL;
  (* Compute the shortest distance between two points *)
END_FUNCTION;

FUNCTION normal(p1, p2, p3 : point) : vector;
  (* Compute normal of a plane given three points on the plane *)
END_FUNCTION;

ENTITY circle;
  centre : point;
  radius : REAL;
  axis   : vector;
DERIVE
  area   : REAL := PI*radius**2;
END_ENTITY;

ENTITY circle_by_points
  SUBTYPE OF (circle);
  p2 : point;
  p3 : point;
DERIVE
  SELF\circle.radius : REAL   := distance(centre,p2);
  SELF\circle.axis   : vector := normal(centre, p2, p3);
WHERE
```

```
  not_coincident : (centre <> p2) AND
                   (p2 <> p3) AND
                   (p3 <> centre);
  is_circle      : distance(centre,p3) =
                   distance(centre,p2);
END_ENTITY;
```

In the subtype, the three defining points (`centre`, `p2`, and `p3`) are explicit attributes while `radius`, `axis`, and `area` are derived attributes. The values of these derived attributes are computed by the expression following the assignment operator. The values of `radius` and `axis` are obtained by means of a function call; the value of `area` is computed in line.

9.2.3.5 Rule inheritance

Every local or global rule that applies to a supertype applies to its subtype(s). Thus, a subtype inherits all the rules of its supertype(s). If a subtype has more than one supertype, the subtype shall inherit all the rules constraining the supertypes.

It is not possible to change or delete any of the rules that are associated with a subtype via rule inheritance but it is possible to add new rules which further constrain the subtype.

Rules and restrictions:

An entity instance shall be constrained by all constraints specified for each of its entity data types. If the constraints specified in two (or more) entity data types conflict, there can be no valid instance which contains those entity data types. An implementation of an *EXPRESS* language parser which claims level 4 conformance shall test for conflicting constraints in entity data types which may coexist in a single instance.

EXAMPLE 52 – In the following, a `graduate` is a `person` who both teaches and takes courses. The `graduate` inherits both attributes and constraints from its supertypes (`teacher` and `student`) together with the attributes and constraints from their mutual supertype (`person`). A `graduate`, unlike a `teacher`, is not allowed to teach graduate level courses.

```
SCHEMA s;
ENTITY person;
  ss_no : INTEGER;
  born  : date;
  ...
DERIVE
  age : INTEGER := years_since(born);
UNIQUE
  un1 : ss_no;
END_ENTITY;

ENTITY teacher
  SUBTYPE OF (person);
  teaches : SET [1:?] OF course;
  ...
WHERE
  old : age >= 21;
END_ENTITY;
```

```
ENTITY student
  SUBTYPE OF (person);
  takes : SET [1:?] OF course;
  ...
WHERE
  young : age >= 5;
END_ENTITY;

ENTITY graduate
  SUBTYPE OF (student, teacher);
  ...
WHERE
  limited : NOT (GRAD_LEVEL IN teaches);
END_ENTITY;

TYPE course = ENUMERATION OF (...., GRAD_LEVEL, ...);
END_TYPE;
...
END_SCHEMA; -- end of schema S
```

NOTE – If a subtype inherits mutually contradictory constraints from its supertypes, there cannot be a conforming instance of that subtype as any instance will violate one of the constraints.

9.2.4 Subtype/supertype constraints

An instance of an entity data type which is a subtype is an instance of each of its supertypes. An instance of an entity data type which is either explicitly or implicitly declared to be a supertype (see 9.2.3.2) may also be an instance of one or more of its subtypes (see G.2).

Syntax:

```
294 subsuper = [ supertype_constraint ] [ subtype_declaration ] .
297 supertype_constraint = abstract_supertype_declaration | supertype_rule .
156 abstract_supertype_declaration = ABSTRACT SUPERTYPE [ subtype_constraint ] .
295 subtype_constraint = OF '(' supertype_expression ')' .
298 supertype_expression = supertype_factor { ANDOR supertype_factor } .
299 supertype_factor = supertype_term { AND supertype_term } .
301 supertype_term = entity_ref | one_of | '(' supertype_expression ')' .
250 one_of = ONEOF '(' supertype_expression { ',' supertype_expression } ')' .
300 supertype_rule = SUPERTYPE subtype_constraint .
```

It is possible to specify constraints on which subtype/supertype graphs may be instantiated. These constraints are specified in the supertype by means of the SUPERTYPE constraint.

Annex B provides the formal approach to determining the potential combinations of subtype/supertype which may be instantiated under the several possible constraints that are described below.

9.2.4.1 Abstract supertypes

EXPRESS allows for the declaration of supertypes that are not intended to be directly instantiated. An entity data type shall include the phrase ABSTRACT SUPERTYPE in a supertype constraint for this purpose. An abstract supertype shall not be instantiated except in conjunction with at least one of its subtypes.

NOTE – This implies that a schema which contains an abstract supertype without any subtypes is incomplete, and cannot be instantiated unless subtypes are declared in a referencing schema.

EXAMPLE 53 – In a transportation model, a vehicle could be represented as an abstract supertype, because all instances of the entity data type are intended to be of its subtypes (i.e., land-based, water-based, etc.). The entity data type for a vehicle must not be independently instantiated.

```
ENTITY vehicle
  ABSTRACT SUPERTYPE;
END_ENTITY;

ENTITY land_based
  SUBTYPE OF (vehicle);
  ...
END_ENTITY;

ENTITY water_based
  SUBTYPE OF (vehicle);
  ...
END_ENTITY;
```

9.2.4.2 ONEOF

The ONEOF constraint states that the elements of the ONEOF list are mutually exclusive. None of the elements may be instantiated with any other element in the list. Each element shall be a supertype expression which may resolve to a single subtype of the entity data type.

Syntax:

```
250 one_of = ONEOF '(' supertype_expression { ',' supertype_expression } ')' .
298 supertype_expression = supertype_factor { ANDOR supertype_factor } .
299 supertype_factor = supertype_term { AND supertype_term } .
301 supertype_term = entity_ref | one_of | '(' supertype_expression ')' .
```

The ONEOF constraint may be combined with the other supertype constraints to enable the writing of complex constraints.

NOTE – The phrase `ONEOF(a,b,c)` reads in natural language as "an instance shall consist of one and only one of the entity data types `a,b,c`."

EXAMPLE 54 – An instance of a supertype may be established through the instantiation of only one of its subtypes. This constraint is declared using the ONEOF constraint. There are many kinds of pet, but no single pet can be simultaneously two or more kinds of pet.

```
ENTITY pet
```

```
  ABSTRACT SUPERTYPE OF (ONEOF(cat,
                               rabbit,
                               dog,
                               ...) );
  name : pet_name;
  ...
END_ENTITY;

ENTITY cat
  SUBTYPE OF (pet);
  ...
END_ENTITY;

ENTITY rabbit
  SUBTYPE OF (pet);
  ...
END_ENTITY;

ENTITY dog
  SUBTYPE OF (pet);
  ...
END_ENTITY;
```

9.2.4.3 ANDOR

If the subtypes are not mutually exclusive, that is, an instance of the supertype may be an instance of more than one of its subtypes, the relationship between the subtypes shall be specified using the ANDOR constraint.

NOTE – The phrase `b ANDOR c` reads in natural language as "an instance shall consist of the types `b` and/or `c`."

EXAMPLE 55 – A person could be an employee who is taking night classes and may therefore be simultaneously both an employee and a student.

```
ENTITY person
  SUPERTYPE OF (employee ANDOR student);
  ...
END_ENTITY;

ENTITY employee
  SUBTYPE OF (person);
  ...
END_ENTITY;

ENTITY student
  SUBTYPE OF (person);
  ...
END_ENTITY;
```

9.2.4.4 AND

If the supertype instances are categorized into multiple groups of mutually exclusive subtypes (i.e., multiple ONEOF groupings) indicating that there is more than one way to completely categorize the supertype, the relationship between those groups shall be specified using the AND constraint. The AND constraint is only used to relate groupings established by other subtype/supertype constraints.

NOTE – The phrase `b AND c` reads in natural language as "an instance shall consist of the types `b` and `c` together."

EXAMPLE 56 – A person could be categorized into being either male or female, and could also be categorized into being either a citizen or an alien.

```
ENTITY person
  SUPERTYPE OF (ONEOF(male,female) AND
                ONEOF(citizen,alien));
  ...
END_ENTITY;

ENTITY male
  SUBTYPE OF (person);
  ...
END_ENTITY;

ENTITY female
  SUBTYPE OF (person);
  ...
END_ENTITY;

ENTITY citizen
  SUBTYPE OF (person);
  ...
END_ENTITY;

ENTITY alien
  SUBTYPE OF (person);
  ...
END_ENTITY;
```

9.2.4.5 Precedence of supertype operators

The evaluation of supertype expressions proceeds from left to right, with the highest precedence operators being evaluated first. Table 8 summarizes the precedence rules for the supertype expression operators. Operators in the same row have the same precedence, and the rows are ordered by decreasing precedence.

EXAMPLE 57 – The following two expressions are not equivalent:

```
ENTITY x
SUPERTYPE OF (a ANDOR b AND c);
```

```
END_ENTITY;

ENTITY x
SUPERTYPE OF ((a ANDOR b) AND c);
END_ENTITY;
```

9.2.4.6 Default constraint between subtypes

If no supertype constraint is mentioned in the declaration of an entity, the subtypes (if any) shall be mutually inclusive, i.e., as if all subtypes were implicitly mentioned in an ANDOR construct.

In the case of a supertype constraint which is specified for a subset of the subtypes of that entity, the constraint shall be as specified for those subtypes mentioned and ANDOR for the other subtypes.

EXAMPLE 58 – The model in example 55 is equivalent to using the default construction, as:

```
ENTITY person
  ...
END_ENTITY;

ENTITY employee
  SUBTYPE OF (person);
  ...
END_ENTITY;

ENTITY student
  SUBTYPE OF (person);
  ...
END_ENTITY;
```

9.2.5 Implicit declarations

When an entity is declared, a constructor is also implicitly declared. The constructor identifier is the same as the entity identifier and the visibility of the constructor declaration is the same as that of the entity declaration.

The constructor, when invoked, shall return a partial complex entity value for that entity data type to the point of invocation. Each attribute in this partial complex entity value is given by the actual parameter passed in the constructor call, if an actual parameter is an entity instance, that entity instance plays the role described by an attribute in the partial complex entity value. The

Table 8 – Supertype expression operator precedence

Precedence	Operators
1	( ) ONEOF
2	AND
3	ANDOR

constructor shall only provide the attributes which are explicit in a particular entity declaration.

Syntax:

```
195 entity_constructor = entity_ref '(' [ expression { ',' expression } ] ')' .
```

When a complex entity instance (an instance of an entity which occurs within a subtype/supertype graph) is constructed, the constructors for each of the component entities shall be combined using the || operator (see 12.10).

Rules and restrictions:

a) The constructor shall have one formal parameter for each explicit attribute declared in that entity data type. This does not include attributes which are inherited from supertypes and are redeclared in that entity data type.

b) The order of the formal parameters shall be identical to the order of declaration of the explicit attributes within the entity.

c) The parameter data type for each of the formal parameters shall be identical to the data type of the corresponding attribute.

d) If an entity contains no explicit attributes, an empty parameter list shall be passed (i.e., the parentheses are always present).

NOTE – This is different from explicitly declared functions.

e) OPTIONAL attributes may be given the value indeterminate (?) when the implicitly defined constructor is invoked. This represents the fact that an explicit value has not been assigned.

f) If within a complex entity instance there exists a subtype which contains derived attributes redeclared from explicit attributes in a supertype, the supertype constructor shall give values for these redeclared attributes. These values are ignored in favour of the derived value.

EXAMPLE 59 – Assuming the following entity declaration:

```
ENTITY point;
  x, y, z : REAL;
END_ENTITY;
```

the implicitly declared constructor for a point can be thought of as:

```
FUNCTION point(x,y,z : REAL):point;
```

the constructor then may be used when assigning values to an instance of this entity data type.

```
CONSTANT
```

```
  origin : point := point(0.0, 0.0, 0.0);
END_CONSTANT;
```

9.2.6 Specialization

A specialization is a more constrained form of an original declaration. The following are the defined specializations:

- a subtype entity is a specialization of any of its supertypes;
- INTEGER and REAL are both specializations of NUMBER;
- INTEGER is a specialization of REAL;
- BOOLEAN is a specialization of LOGICAL;
- `LIST OF UNIQUE item` is a specialization of `LIST OF item`;
- `ARRAY OF UNIQUE item` is a specialization of `ARRAY OF item`;
- `ARRAY OF item` is a specialization of `ARRAY OF OPTIONAL item`;
- `SET OF item` is a specialization of `BAG OF item`;
- letting `AGG` stand for one of ARRAY, BAG, LIST or SET then `AGG OF item` is a specialization of `AGG OF original` provided that `item` is a specialization of `original`;
- letting `AGG` stand for one of BAG, LIST or SET then `AGG [b:t]` is a specialization of `AGG [l:u]` provided that $b \leq t$ and $l \leq b \leq u$ and $l \leq t \leq u$;
- letting `BSR` stand for one of the data types BINARY, STRING or REAL then `BSR (length)` is a specialization of `BSR`;
- `BSR (short)` is a specialization of `BSR (long)` provided that `short` is less than `long`;
- a BINARY which uses the keyword FIXED is a specialization of variable length BINARY;
- a STRING which uses the keyword FIXED is a specialization of variable length STRING;
- a defined data type is a specialization of the underlying data type used to declare the defined data type.

9.3 Schema

A SCHEMA declaration defines a common scope for a collection of related entity and other data type declarations.

EXAMPLE 60 – Geometry may be the name of a schema that contains declarations of points, curves, surfaces and other related data types.

The order in which declarations appear within a schema declaration is not significant.

Declarations in one schema may be made visible within the scope of another schema via an interface specification as described in clause 11.

Syntax:

```
281 schema_decl = SCHEMA schema_id ';' schema_body END_SCHEMA ';' .
280 schema_body = { interface_specification } [ constant_decl ]
                  { declaration | rule_decl } .
228 interface_specification = reference_clause | use_clause .
189 declaration = entity_decl | function_decl | procedure_decl | type_decl .
```

9.4 Constant

A constant declaration is used to declare named constants. The scope of a constant identifier shall be the function, procedure, rule or schema in which the constant declaration occurs. A named constant appearing in a CONSTANT declaration shall have an explicit initialization the value of which is computed by evaluating the expression. A named constant may appear in the declaration of another named constant.

Syntax:

```
185 constant_decl = CONSTANT constant_body { constant_body } END_CONSTANT ';' .
184 constant_body = constant_id ':' base_type ':=' expression ';' .
171 base_type = aggregation_types | simple_types | named_types .
```

Rules and restrictions:

a) The value of a constant shall not be modified after initialization.

b) Any occurrence of a named constant outside the constant declaration shall be equivalent to an occurrence of the initial value itself.

c) The `expression` shall return a value of the specified base type.

EXAMPLE 61 – The following are valid constant declarations:

```
CONSTANT
  thousand : NUMBER := 1000;
  million  : NUMBER := thousand**2;
  origin   : point := point(0.0, 0.0, 0.0);
END_CONSTANT;
```

9.5 Algorithms

An algorithm is a sequence of statements that produces some desired end state. The two kinds of algorithms that can be specified are functions and procedures.

Formal parameters define the input to an algorithm. When an algorithm is called, actual parameters supply actual values or instances. The actual parameters shall agree in type, order, and number with the formal parameters.

Declarations local to the algorithm, if required, are given following the header. These declarations can be types, local variables, other algorithms, etc., as needed.

The body of the algorithm follows the local declarations.

9.5.1 Function

A function is an algorithm which operates on parameters and that produces a single resultant value of a specific data type. An invocation of a function (see 12.8) in an expression evaluates to the resultant value at the point of invocation.

A function shall be terminated by the execution of a RETURN statement. The value of the expression, associated with the RETURN statement, defines the result produced by the function call.

Syntax:

```
208 function_decl = function_head [ algorithm_head ] stmt { stmt } END_FUNCTION ';' .
 209 function_head = FUNCTION function_id [ '(' formal_parameter
                     { ';' formal_parameter } ')' ] ':' parameter_type ';' .
 206 formal_parameter = parameter_id { ',' parameter_id } ':' parameter_type .
 253 parameter_type = generalized_types | named_types | simple_types .
 163 algorithm_head = { declaration } [ constant_decl ] [ local_decl ] .
 189 declaration = entity_decl | function_decl | procedure_decl | type_decl .
```

Rules and restrictions:

a) A FUNCTION shall specify a RETURN statement in each of the possible paths a process may take when that function is invoked.

b) Each RETURN statement within the scope of the function shall specify an expression which evaluates to the value to be returned to the point of invocation.

c) The expression specified in each RETURN statement shall be assignment compatible with the declared return type of the function.

d) Functions do not have side-effects. Since the formal parameters to a FUNCTION shall not be specified to be VAR, changes to these parameters within the function are not reflected to the point of invocation.

e) Functions may modify local variables or parameters which are declared in an outer scope, i.e., if the current function is declared within the `algorithm_head` of a FUNCTION, PROCEDURE or RULE.

9.5.2 Procedure

A procedure is an algorithm that receives parameters from the point of invocation and operates on them in some manner to produce the desired end state. Changes to the parameters within a procedure are only reflected to the point of invocation when the formal parameter is preceded by the VAR keyword.

Syntax:

```
258 procedure_decl = procedure_head [ algorithm_head ] { stmt } END_PROCEDURE ';' .
259 procedure_head = PROCEDURE procedure_id [ '(' [ VAR ] formal_parameter
                    { ';' [ VAR ] formal_parameter } ')' ] ';' .
206 formal_parameter = parameter_id { ',' parameter_id } ':' parameter_type .
253 parameter_type = generalized_types | named_types | simple_types .
163 algorithm_head = { declaration } [ constant_decl ] [ local_decl ] .
189 declaration = entity_decl | function_decl | procedure_decl | type_decl .
```

Rules and restrictions:

Procedures may modify local variables or parameters which are declared in an outer scope, i.e., if the current procedure is declared within the `algorithm_head` of a FUNCTION, PROCEDURE or RULE.

9.5.3 Parameters

A function or procedure can have formal parameters. Each formal parameter specifies a name and a parameter type. The name is an identifier which shall be unique within the scope of the function or procedure. A formal parameter to a procedure may also be declared as VAR (variable), which means that, if the parameter is changed within the procedure, the change shall be propagated to the point of invocation. Parameters not declared as VAR can be changed also, but the change will not be apparent when control is returned to the caller.

Syntax:

```
206 formal_parameter = parameter_id { ',' parameter_id } ':' parameter_type .
253 parameter_type = generalized_types | named_types | simple_types .
```

EXAMPLE 62 – The following declarations indicate how formal parameters may be declared.

```
FUNCTION dist(p1, p2 : point) : REAL ;
...
PROCEDURE midpt(p1, p2 : point; VAR result : point) ;
```

The generalization data types (AGGREGATE and GENERIC) are used to allow generalization of the data types used to represent the formal parameters of functions and procedures. General aggregation data types may also be used to allow a generalization of the underlying data types allowed for the specific aggregation data types.

9.5.3.1 Aggregate data type

An AGGREGATE data type is a generalization of all aggregation data types.

When a procedure or function which has a formal parameter defined to be an aggregate data type is invoked, the actual parameter passed shall be an ARRAY, BAG, LIST or SET. The operations that can be performed shall then depend on the data type of the actual parameter.

Type labels may be used to ensure that two or more parameters passed are of the same data type, or that the return data type is the same as one of the passed parameters, irrespective of the actual data types passed (see 9.5.3.3).

Syntax:

```
161 aggregate_type = AGGREGATE [ ':' type_label ] OF parameter_type .
306 type_label = type_label_id | type_label_ref .
253 parameter_type = generalized_types | named_types | simple_types .
```

Rules and restrictions:

a) An AGGREGATE data type shall only be used as a formal parameter type of a function or procedure, or as specified in rule (b).

b) An AGGREGATE data type may also be used as the result type of a function, or as the type of a local variable within a function or procedure. Type labels references are required for this usage and shall refer to type labels declared by the formal parameters (see 9.5.3.3).

EXAMPLE 63 – A function is written to accept an aggregate of numbers. It shall return the same type of aggregate passed containing the scaled numbers.

```
FUNCTION scale(input:AGGREGATE:intype OF NUMBER;
               scalar:NUMBER):AGGREGATE:intype OF NUMBER;
  LOCAL
    result : AGGREGATE:intype OF NUMBER;
  END_LOCAL;

  REPEAT i := LOINDEX(input) TO HIINDEX(input);
    result[i] := scalar * input[i];
  END_REPEAT;

  RETURN(result);

END_FUNCTION;
```

9.5.3.2 Generic data type

A GENERIC data type is a generalization of all other data types.

When a procedure or function is invoked with a generic parameter, the actual parameter passed may not be of GENERIC data type. The operations that can be performed depend on the data type of the actual parameter.

Type labels may be used to ensure that two or more parameters passed are of the same data

type, or that the return data type is the same as one of the passed parameters, irrespective of the actual data types passed (see 9.5.3.3).

Syntax:

```
218 generic_type = GENERIC [ ':' type_label ] .
306 type_label = type_label_id | type_label_ref .
```

Rules and restrictions:

a) A GENERIC data type shall only be used as a formal parameter type of a function or procedure, or as specified in rule (b).

b) A GENERIC data type may also be used as the result type of a function, or as the type of a local variable within a function or procedure. Type labels references are required for this usage and shall refer to type labels declared by the formal parameters (see 9.5.3.3).

EXAMPLE 64 – This example shows a generic function that adds numbers or vectors.

```
FUNCTION add(a,b:GENERIC:intype):GENERIC:intype;
  LOCAL
    nr : NUMBER; -- integer or real
    vr : vector;
  END_LOCAL;

  IF ('NUMBER' IN TYPEOF(a)) AND ('NUMBER' IN TYPEOF(b)) THEN
       nr := a+b;
       RETURN(nr);
  ELSE
    IF ('THIS_SCHEMA.VECTOR' IN TYPEOF(a)) AND
       ('THIS_SCHEMA.VECTOR' IN TYPEOF(b)) THEN
      vr := vector(a.i + b.i,
                   a.j + b.j,
                   a.k + b.k);
      RETURN(vr);
    END_IF;
  END_IF;
  RETURN (?); -- if we receive input which is invalid then return a no value
END_FUNCTION;
```

9.5.3.3 Type labels

Type labels shall be used to relate the data type of an actual parameter at invocation to the data types of other actual parameters, local variables, or the return type of a function. Type labels are declared for AGGREGATE and GENERIC data types within the formal parameter declaration of a function or procedure and may be referenced by AGGREGATE or GENERIC data types in the formal parameter declaration, local variable declaration or the declaration of the returned data type of a FUNCTION.

Syntax:

```
306 type_label = type_label_id | type_label_ref .
```

Rules and restrictions:

a) The first occurrence of a type label in a formal parameter declaration declares that type label, subsequent occurrences of that type label are references to the first.

b) The parameters passed to a function or procedure which use a reference to a type label shall be compatible with the data type of the parameter passed which declares the type label.

c) The data types of local variables and return types of functions which refer via a type label to a parameter data type shall be identical to parameter data type which declares the type label.

EXAMPLE 65 – This example indicates how type labels may be used when specifying a function, and the resulting type compatibility checks used when invoking the function.

```
ENTITY a;
...
END_ENTITY;

ENTITY b SUBTYPE OF (a);
...
END_ENTITY;

ENTITY c SUBTYPE OF (b);
...
END_ENTITY;

...

FUNCTION test ( p1 : GENERIC:x; p2 : GENERIC:x):GENERIC:x;
...     --                    ^               ^          ^
...     --                  declaration     reference  reference
END_FUNCTION;

...

LOCAL
  v_a : a := a(...);
  v_b : b := a(...)||b(...);   -- || operator is described in 12.11
  v_c : c := a(...)||b(...)||c(...);
  v_x : b;
END_LOCAL;

v_x := test(v_b, v_a);  -- invalid v_a is not compatible with type b.
v_x := test(v_a, v_b);  -- invalid assignment, function will return a type a
```

Further examples of type label usage can be seen in clause 15.

9.5.3.4 General aggregation data types

General aggregation data types form part of the class of types called generalized data types. They represent a specific aggregation data type (ARRAY, BAG, LIST and SET) with a relaxing of the constraints which would normally be applied when specifying the aggregation data type (i.e., a `general_list_type` is a generalization of a `list_type`).

Syntax:

```
212 general_aggregation_types = general_array_type | general_bag_type |
                                general_list_type | general_set_type .
213 general_array_type = ARRAY [ bound_spec ] OF [ OPTIONAL ] [ UNIQUE ]
                         parameter_type .
176 bound_spec = '[' bound_1 ':' bound_2 ']' .
174 bound_1 = numeric_expression .
175 bound_2 = numeric_expression .
253 parameter_type = generalized_types | named_types | simple_types .
214 general_bag_type = BAG [ bound_spec ] OF parameter_type .
215 general_list_type = LIST [ bound_spec ] OF [ UNIQUE ] parameter_type .
217 general_set_type = SET [ bound_spec ] OF parameter_type .
```

When used as the data type of a formal parameter, general aggregation data types allow the following to be actual parameters of functions and procedures:

– arrays with no regard to the range of the index values. This is done by not specifying a bound spec for the array in the formal parameter specification;

NOTE – The functions HIINDEX and LOINDEX should be used in the algorithmic part to determine the actual indexing system of the array.

– aggregations in which the underlying type may be a GENERIC, AGGREGATE or general aggregation data type.

EXAMPLE 66 – This example indicates how a SET may be written in a formal parameter declaration, this could not be written in an attribute declaration since the underlying type for SET does not include GENERIC.

```
FUNCTION dimensions(input:SET [2:3] OF GENERIC):INTEGER;
```

9.5.4 Local variables

Variables local to an algorithm are declared after the LOCAL keyword. A local variable is only visible within the scope of the algorithm in which it is declared. Local variables may be assigned values and may participate in expressions.

Syntax:

```
239 local_decl = LOCAL local_variable { local_variable } END_LOCAL ';' .
240 local_variable = variable_id { ',' variable_id } ':' parameter_type
                    [ ':=' expression ] ';' .
253 parameter_type = generalized_types | named_types | simple_types .
```

Initialization of local variables

When an algorithm is invoked all local variables have a value of indeterminate (?) unless an initializer is explicitly given. When an initializer is given, the initialization value is assigned to the local variable upon entry to the algorithm.

EXAMPLE 67 – The variable `r_result` is initialized with a value of `0.0`

```
LOCAL
  r_result : REAL := 0.0;
  i_result : INTEGER;
END_LOCAl;
...
EXISTS(r_result) -- TRUE
EXISTS(i_result) -- FALSE assuming there has been no value assigned
```

9.6 Rule

Rules permit the definition of constraints that apply to one or more entity data types within the scope of a schema. Local rules (i.e., the uniqueness constraints and domain rules in an entity declaration) declare constraints that apply individually to every instance of an entity data type. A RULE declaration permits the definition of constraints that apply collectively to the entire domain of an entity data type, or to instances of more than one entity data type. One application of a RULE is to constrain the values of attributes that exist in different entities in a coordinated manner.

A rule declaration names the rule and specifies the entities affected by it.

Syntax:

```
277 rule_decl = rule_head [ algorithm_head ] { stmt } where_clause END_RULE ';' .
278 rule_head = RULE rule_id FOR '(' entity_ref { ',' entity_ref } ')' ';' .
163 algorithm_head = { declaration } [ constant_decl ] [ local_decl ] .
189 declaration = entity_decl | function_decl | procedure_decl | type_decl .
```

The body of a rule consists of local declarations, executable statements and domain rules. The end state of a rule indicates whether or not some global constraint is satisfied. A rule is evaluated by executing the statements and then evaluating each of the domain rules. If a rule is violated for the set of instances of the entity data types passed as parameters, the instances do not conform to the *EXPRESS* schema.

Rules and restrictions:

a) Each domain rule shall evaluate to either a LOGICAL value or indeterminate (?).

b) The expression is asserted when it evaluates to a value of TRUE; it is violated when it evaluates to a value of FALSE; and it is neither violated nor asserted when the expression evaluates to a indeterminate (?) or an UNKNOWN value.

c) No domain rule shall be violated for a valid collection of entity instances of the entity data types specified in the header of the rule.

EXAMPLE 68 – The following rule demands that there are equal numbers of points in the first and seventh octants.

```
RULE point_match FOR (point);
LOCAL
  first_oct ,
  seventh_oct : SET OF POINT := [];  -- empty set of point (see 12.9)
END_LOCAL
  first_oct := QUERY(temp <* point | (temp.x > 0) AND
                                     (temp.y > 0) AND
                                     (temp.z > 0) );
  seventh_oct := QUERY(temp <* point | (temp.x < 0) AND
                                       (temp.y < 0) AND
                                       (temp.z < 0) );
WHERE
  SIZEOF(first_oct) = SIZEOF(seventh_oct);
END_RULE;
```

EXAMPLE 69 – A RULE may be used to specify joint value uniquenes for entity attributes.

```
ENTITY b;
  a1 : c;
  a2 : d;
  a3 : f;
UNIQUE
  ur1 : a1, a2;
END_ENTITY;
```

The joint uniqueness constraint in b applies to instances of c and d. The following RULE further constrains the joint uniqueness to be value-based.

```
RULE vu FOR (b);
  ENTITY temp;
    a1 : c;
    a2 : d;
  END_ENTITY;
LOCAL
  s : SET OF temp := [];
END_LOCAL;
REPEAT i := 1 TO SIZEOF(b);
  s := s + temp(b[i].a1, b[i].a2);
END_REPEAT;
WHERE
  wr1 : VALUE_UNIQUE(s);
```

```
END_RULE;
```

Implicit declaration

Within a RULE each `population` is implicitly declared to be a local variable which contains the set of all instances of the named entity in the domain; i.e., the set of entity instances governed by the rule.

Syntax:

```
254 population = entity_ref .
```

Rules and restrictions:

References to a particular `population` may only be made in a global rule which references that entity data type in the header of the rule.

EXAMPLE 70 – Assuming the following declaration:

```
RULE coincident FOR (point);
```

the implicitly declared variable would be:

```
LOCAL
  point : SET OF point;
END_LOCAL;
```

10 Scope and visibility

An *EXPRESS* declaration creates an identifier which can be used to reference the declared item in other parts of the schema (or in other schemas). Some *EXPRESS* constructs implicitly declare *EXPRESS* items, attaching identifiers to them. In those areas where an identifier for a declared item may be referenced, the declared item is said to be visible. An item may only be referenced where its identifier is visible. For the rules of visibility, see 10.2. For further information on referring to items using their identifiers, see 12.7.

Certain *EXPRESS* items define a region (block) of text called the scope of item. This scope limits the visibility of identifiers declared within it. Scopes can be nested; that is, an *EXPRESS* item which establishes a scope may be included within the scope of another item. There are constraints on which items may appear within a particular *EXPRESS* item's scope. These constraints are usually enforced by the syntax of *EXPRESS* (see annex A).

For each of the items specified in table 9 the following subclauses specify the limits of the scope defined, if any, and the visibility of the declared identifier both in general terms and with specific details.

Table 9 – Scope and identifier defining items

Item	Scope	Identifier
alias statement	•	•[1]
attribute		•
constant		•
enumeration		•
entity	•	•
function	•	•
parameter		•
procedure	•	•
query expression	•	•[1]
repeat statement	•	•[1,2]
rule	•	•[3]
rule label		•
schema	•	•
type	•	•
type label		•
variable		•

NOTES

1 – The identifier is an implicitly declared variable within the defined scope of the declaration.

2 – The variable is only implicitly declared when an increment control is specified.

3 – An implicit variable declaration is made for all entities which are constrained by the rule.

10.1 Scope rules

The following are the general rules which are applicable to all forms of scope definition allowed within the *EXPRESS* language; see table 9 for the list of items which define scopes.

Rules and restrictions:

a) All declarations shall exist within a scope.

b) Within a single scope an identifier may be declared, or explicitly interfaced (see clause 11), once only. An entity or type identifier which has been explicitly interfaced into the current schema via two, or more, routes which share the same initial declaration is counted only once.

c) The scopes shall be correctly nested, i.e., scopes shall not overlap. (This is forced by the syntax of the language.)

A maximum permitted depth of nesting is not specified by this part of ISO 10303 but implementations of *EXPRESS* parsers may specify a maximum depth of scope nesting.

10.2 Visibility rules

The visibility rules for identifiers are described below. See table 9 for the list of *EXPRESS* items which declare identifiers. The visibility rules for named data type identifiers are slightly different from those for other identifiers; these differences are described in 10.2.2.

10.2.1 General rules of visibility

The following are the general rules which are applicable to all identifiers except the named data type identifiers, for which rule (d) does not apply.

Rules and restrictions:

a) An identifier is visible in the scope in which it is declared. This scope is called the local scope of the identifier.

b) If an identifier is visible in a particular scope, it is also visible in all scopes defined within that scope, subject to rule (d).

c) An identifier is not visible in any scope outside its local scope, subject to rule (f).

d) When an identifier i visible in a scope P is re-declared in some inner scope Q enclosed within P, only the i declared in scope Q is visible in Q and any scopes declared within Q. The i declared in scope P is visible in P and in any inner scopes which do not re-declare i.

e) The built-in constants, functions, procedures and types of *EXPRESS* are considered to be declared in an imaginary universal scope. All schemas are nested within this scope. The identifiers which refer to the built-in constants, functions, procedures, types of *EXPRESS* and schemas are visible in all scopes defined by *EXPRESS*.

f) Enumeration item identifiers declared within the scope of a defined data type are visible in the next outer scope, unless the next outer scope contains a declaration of the same identifier for some other item.

NOTE – If the next outer scope contains a declaration of the same identifier the enumeration items are still accessible but have to be prefixed by the defined data type identifier (see 12.7.2).

g) Declarations in one schema are made visible to items in another schema by the interface specification (see clause 11).

EXAMPLE 71 – The following schema shows examples of identifiers and references which are allowed according to the above rules.

```
SCHEMA example;

  CONSTANT
    b : INTEGER := 1 ;
    c : BOOLEAN := TRUE ;
  END_CONSTANT;

  TYPE enum = ENUMERATION OF ( e, f, g );
  END_TYPE;

  ENTITY entity1;
    a : INTEGER;
  WHERE
    wr1: a > 0 ;              -- obeys rule (a): a is visible in local scope
    wr2: a <> b ;             -- obeys rule (b): b is visible from outer scope
  END_ENTITY;

  ENTITY entity2;
    c : REAL;                 -- obeys rule (c) constant c invisible here
  END_ENTITY;

  ENTITY d;
    attr1 : INTEGER;
    attr2 : enum;
  WHERE
    d: ODD(attr1);             -- obeys rule (d) ODD is visible anywhere
    wr: attr2 <> e;           -- obeys rule (e) e is visible outside the scope
  END_ENTITY;                 --                 defined by the type enum

END_SCHEMA;
```

10.2.2 Named data type identifier visibility rules

With one exception, named data type identifiers obey the same visibility rules as other identifiers. The exception is to visibility rule (d). An entity or defined data type identifier i declared in a scope P remains visible in an inner scope Q even if it is redeclared in Q, provided that either:

a) The scope Q is defined by an entity declaration, and i is declared as an attribute in that scope, or

b) The scope Q is defined by a function, procedure or rule declaration, and i is declared as a formal parameter or variable in that scope.

EXAMPLE 72 – In `entity1`, `d` refers to both an entity data type and an attribute.

```
SCHEMA example;

  ENTITY d;
    attr1 : REAL;
  END_ENTITY;

  ENTITY entity1;
    d : d;                    -- d in this scope is both an entity
  END_ENTITY;                 --  and an attribute.

END_SCHEMA;
```

10.3 Explicit item rules

The following clauses provide more detail on how the general scoping and visibility rules apply to the various *EXPRESS* items.

10.3.1 Alias statement

See 13.2 for the definition of the ALIAS statement.

Visibility : The implicitly declared identifier in an alias statement is visible in the scope defined by that alias statement.

Scope : An alias statement defines a new scope. This scope extends from the keyword ALIAS to the keyword END_ALIAS which terminates that alias statement.

Declarations : The following *EXPRESS* items may declare identifiers within the scope of an alias statement:

- alias statement;
- query expression;
- repeat statement.

10.3.2 Attribute

Visibility : An attribute identifier is visible in the scope of the entity in which it is declared and all subtypes of that entity.

10.3.3 Constant

Visibility : A constant identifier is visible in the scope of the function, procedure, rule or schema in which it is declared.

10.3.4 Enumeration item

Visibility : An enumeration item identifier is visible in the scope of the function, procedure, rule or schema in which its type is declared. This is the exception to the visibility rule (f) of 10.2.1. The identifier shall not be declared for any other purpose in this scope, except by another enumeration data type declaration in the same scope. If the same identifier is declared by two enumeration data types as an enumeration item, a reference to either enumeration item must be prefixed with the data type identifier in order to ensure that the reference is unambiguous (see 8.4.1).

10.3.5 Entity

Visibility : An entity identifier is visible in the scope of the function, procedure, rule or schema in which it is declared. An entity identifier remains visible, under the conditions defined in 10.2.2, within inner scopes which redeclare that identifier.

Scope : An entity declaration defines a new scope. This scope extends from the keyword ENTITY to the keyword END_ENTITY which terminates that entity declaration. Attributes declared in a supertype of an entity are visible in the subtype entity through inheritance.

NOTE – The scope of the subtype entity is not considered to be nested within the scope of the supertype.

Declarations : The following *EXPRESS* items may declare identifiers within the scope of an entity declaration:

- attribute (explicit, derived and inverse);
- rule label (unique and domain rules);
- query expression (within derived attributes and domain rules).

EXAMPLE 73 – The attribute identifiers `batt` in the two entities do not clash as they are declared in two different scopes.

```
ENTITY entity1;
  aatt : INTEGER;
  batt : INTEGER;
END_ENTITY;

ENTITY entity2;
  a    : entity1;
  batt : INTEGER;
END_ENTITY;
```

EXAMPLE 74 – The following specification is illegal because the attribute identifier `aatt` is both inherited and declared within the scope of `illegal` (see 9.2.3.3). The rule label `lab` in the two entities do not clash since they are in separate scopes; a valid instance of `illegal`, ignoring the error with attribute `aatt`, obeys both domain rules.

```
ENTITY may_be_ok;
  quantity : REAL;
  aatt : REAL;
WHERE
  lab : quantity >= 0.0;
END_ENTITY;

ENTITY illegal
  SUBTYPE OF (may_be_ok);
  aatt : INTEGER;
  batt : INTEGER;
WHERE
  lab : batt < 0;
END_ENTITY;
```

10.3.6 Function

Visibility : A function identifier is visible in the scope of the function, procedure, rule or schema in which it is declared.

Scope : A function declaration defines a new scope. This scope extends from the keyword FUNCTION to the keyword END_FUNCTION which terminates that function declaration.

Declarations : The following *EXPRESS* items may declare identifiers within the scope of a function declaration:

- alias statement;
- constant;
- entity;
- enumeration;
- function;
- parameter;
- procedure;
- query expression;
- return statement;
- type;

– type label;

– variable.

EXAMPLE 75 – The following is illegal, as the formal parameter identifier `parm` is also used as the identifier of a local variable.

```
FUNCTION illegal(parm : REAL) : LOGICAL;
LOCAL
  parm : STRING;
END_LOCAL;
  ...
END_FUNCTION;
```

10.3.7 Parameter

Visibility : A formal parameter identifier is visible in the scope of the function or procedure in which it is declared.

10.3.8 Procedure

Visibility : A procedure identifier is visible in the scope of the function, procedure, rule or schema in which it is declared.

Scope : A procedure declaration defines a new scope. This scope extends from the keyword PROCEDURE to the keyword END_PROCEDURE which terminates that procedure declaration.

Declarations : The following *EXPRESS* items may declare identifiers within the scope of a procedure declaration:

– alias statement;

– constant;

– entity;

– enumeration;

– function;

– parameter;

– procedure;

– query expression;

– return statement;

– type;

– type label;

– variable.

10.3.9 Query expression

See 12.6.7 for the definition of the QUERY expression.

Visibility : The implicitly declared identifier in a query expression is visible in the scope defined by that query expression.

Scope : A query expression defines a new scope. This scope extends from the opening parenthesis '(' after the keyword QUERY to the closing parenthesis ')' which terminates that query expression.

Declarations : The following *EXPRESS* items may declare identifiers within the scope of a query expression:

– query expression.

10.3.10 Repeat statement

See 13.9 for the definition of the REPEAT statement.

Visibility : The implicitly declared identifier in an increment-controlled repeat statement is visible within the scope of that repeat statement.

Scope : A repeat statement defines a new scope. This scope extends from the keyword REPEAT to the keyword END_REPEAT which terminates that repeat statement.

Declarations : The following *EXPRESS* items may declare identifiers within the scope of a repeat statement:

– alias statement;

– query expression;

– repeat statement.

10.3.11 Rule

Visibility : A rule identifier is visible in the scope of the schema in which it is declared.

NOTE – The rule identifier is of use only to an implementation. *EXPRESS* provides no mechanism for referencing rule identifiers.

Scope : A rule declaration defines a new scope. This scope extends from the keyword RULE to the keyword END_RULE which terminates that rule declaration.

Declarations : The following *EXPRESS* items may declare identifiers within the scope of a rule declaration:

– alias statement;

- constant;
- entity;
- enumeration;
- function;
- parameter;
- procedure;
- query expression;
- return statement;
- rule label;
- type;
- type label;
- variable.

EXAMPLE 76 – The following is illegal, since the identifier `point` of the entity affected by the rule, which is implicitly declared as a variable inside the rule, is also explicitly declared as a local variable.

```
RULE illegal FOR (point);
LOCAL
  point : STRING;
END_LOCAL;
  ...
END_RULE;
```

10.3.12 Rule label

Visibility : A rule label is visible in the scope of the entity, rule or type in which it is declared.

NOTE – The rule label is only of use to an implementation. *EXPRESS* provides no mechanism for referencing rule labels.

10.3.13 Schema

Visibility : A schema identifier is visible to all other schemas.

NOTE – A conformant implementation may provide a scoping mechanism which allows a collection of schemas to be treated as a scope.

Scope : A schema declaration defines a new scope. This scope extends from the keyword SCHEMA to the keyword END_SCHEMA which terminates that schema declaration.

Declarations : The following *EXPRESS* items may declare identifiers within the scope of a schema declaration:

- constant;
- entity;
- enumeration;
- function;
- procedure;
- rule;
- type.

EXAMPLE 77 – The following schema is illegal on two counts. The identifier `adef` has been imported via the USE clause but has been redeclared as the name of a type. In the second case, the name `fdef` has been used as the identifier for two declarations (which happen to be an entity and a function, although the item type is irrelevant).

```
SCHEMA incorrect;
USE FROM another_schema (adef);

  FUNCTION fdef(parm : NUMBER) : INTEGER;
  ...
  END_FUNCTION;

  TYPE adef = STRING;
  END_TYPE;

  ENTITY fdef;
  ...
  END_ENTITY;

END_SCHEMA;
```

10.3.14 Type

Visibility : A type identifier is visible in the scope of the function, procedure, rule or schema in which it is declared. A type identifier remains visible, under the conditions defined in 10.2.2, within inner scopes which redeclare that identifier.

Scope : A type declaration creates a new scope. This scope extends from the keyword TYPE to the keyword END_TYPE which terminates that type declaration.

Declarations : The following *EXPRESS* items may declare identifiers within the scope of a type declaration:

- enumeration;
- rule label (domain rule);
- query expression (within domain rule).

10.3.15 Type label

Visibility : A type label is visible in the scope of the function or procedure in which it is declared. It is implicitly declared by its first appearance in the scope, which shall be in the formal parameter specification. A type label thus declared may be referenced elsewhere in the formal parameter specification or in the local declarations of the function or procedure. If it is declared in a function, the type label may be referenced in the function's result type specification.

10.3.16 Variable

Visibility : A variable identifier is visible in the scope of the function, procedure or rule in which it is declared.

11 Interface specification

This clause specifies the constructs which enable items declared in one schema to be visible in another. There are two interface specifications (USE and REFERENCE), both of which enable item visibility. The USE specification allows items declared in one schema to be independently instantiated in the schema specifying the USE construct.

An entity instance is independent if it does not play the role described by an attribute of any other entity instance, i.e., ROLESOF (see 15.20) for an independent entity instance will return an empty set. An entity data type which was either declared locally within or USE'd by the schema may be instantiated independently or play the role described by an attribute of an entity within the schema.

An entity which is either explicitly REFERENCE'd or implicitly interfaced shall only be instantiated to play the role described by an attribute of an instantiation of an entity in the schema.

Syntax:

```
228 interface_specification = reference_clause | use_clause .
```

A foreign declaration is any declaration (such as an entity) which appears in a foreign schema (any schema other than the current schema).

A further distinction between the two forms of interface is that the USE specification applies to only named data types (entity data types and defined data types), while the REFERENCE

specification applies to all declarations except rules and schemas.

A foreign *EXPRESS* item may be given a new name in the current schema. The *EXPRESS* item shall be referred to in the current schema by the new name if given following the AS keyword.

11.1 Use interface specification

An entity data type or defined data type declared in a foreign schema is made usable by way of a USE specification. The USE specification gives the name of the foreign schema and optionally the names of entity data types or defined data types declared therein. If there are no `named_types` specified, all of the named data types declared within or USE'd by the foreign schema are treated as if declared locally.

Syntax:

```
313 use_clause = USE FROM schema_ref [ '(' named_type_or_rename
                 { ',' named_type_or_rename } ')' ] ';' .
246 named_type_or_rename = named_types [ AS ( entity_id | type_id ) ] .
```

11.2 Reference interface specification

A REFERENCE specification enables the following *EXPRESS* items, declared in a foreign schema, to be visible in the current schema:

- Constant;
- Entity;
- Function;
- Procedure;
- Type.

The REFERENCE specification gives the name of the foreign schema, and optionally the names of *EXPRESS* items declared therein. If there are no names specified, all the *EXPRESS* items declared in or USE'd by the foreign schema are visible within the current schema.

Syntax:

```
267 reference_clause = REFERENCE FROM schema_ref [ '(' resource_or_rename
                       { ',' resource_or_rename } ')' ] ';' .
274 resource_or_rename = resource_ref [ AS rename_id ] .
275 resource_ref = constant_ref | entity_ref | function_ref | procedure_ref |
                  type_ref .
270 rename_id = constant_id | entity_id | function_id | procedure_id | type_id .
```

REFERENCE'd foreign declarations are not treated as local declarations, and therefore cannot be independently instantiated, but may be instantiated to play the role described by an attribute of an entity in the current schema.

11.3 The interaction of use and reference

If an entity data type or defined data type is both USE'd and REFERENCE'd into the current schema, the USE specification takes precedence.

EXAMPLE 78 – The statements

```
USE FROM s1 (a1);

REFERENCE FROM s1 (a1);
```

treat `a1` as a local declaration.

When a named data type is USE'd into the current schema, that named data type may be USE'd or REFERENCE'd from the current schema by another schema (i.e. USE specifications may be chained between schemas).

EXAMPLE 79 – Given the following two schema declarations:

```
SCHEMA s1;
  ENTITY e1;
  END_ENTITY;
END_SCHEMA;

SCHEMA s2;
USE FROM s1 (e1 AS e2);
END_SCHEMA;
```

the following specifications are equivalent.

```
SCHEMA s3;                          SCHEMA s3;
USE FROM s1 (e1 AS e2);             USE FROM s2 (e2);
END_SCHEMA;                         END_SCHEMA;
```

Since REFERENCE does not treat the *EXPRESS* items REFERENCE'd as local it is not possible to chain REFERENCE's.

11.4 Implicit interfaces

A foreign declaration may refer to identifiers which are not visible to the current schema. Those *EXPRESS* items referred to implicitly are required for a full understanding of the current schema, but they are not visible to *EXPRESS* items declared in the current schema. Each implicitly interfaced item may in turn refer to other *EXPRESS* items which are not visible in the current schema; those *EXPRESS* items also are required for a full understanding of the current schema.

EXAMPLE 80 – Implicitly interfaced items, and chaining of implicit interfaces.

```
SCHEMA s1;

  TYPE t1 = REAL;
```

```
  END_TYPE;

  ENTITY e1;
    a  : t1;
  END_ENTITY;

  ENTITY e2;
    a1 : e1;
  END_ENTITY;
END_SCHEMA:

SCHEMA s2;
  REFERENCE FROM s1 (e2);

  ENTITY e3;
    a3: e2;
  END_ENTITY;
END_SCHEMA;
```

The entity `e2` is used as the data type of the attribute `a3`. Since `e2` requires `e1` in its definition, `e1` is implicitly interfaced by schema `s2`. However since `e1` has not been explicitly interfaced in `s2`, `e1` cannot be specifically mentioned within `s2`. Similarly `e1` requires `t1` in its definition; `t1` is therefore implicitly interfaced by schema `s2`.

In the following subclauses the word interfaced will be used to mean USE'd, REFERENCE'd or implicitly interfaced.

11.4.1 Constant interfaces

When a constant is interfaced, the following are implicitly interfaced:

- any defined data types used in the declaration of the interfaced constant;
- any entity data types used in the declaration of the interfaced constant;
- any constants used in the declaration of the interfaced constant;
- any functions used in the declaration of the interfaced constant.

11.4.2 Defined data type interfaces

When a defined data type is interfaced, the following are implicitly interfaced:

- any defined data types used in the declaration of the interfaced type, except if the interfaced type is a SELECT type; none of the selectable items are implicitly interfaced as a result of this interface;
- any constants or functions used in the declaration of the representation of the interfaced defined data type;

– any constants or functions used within the domain rules of the interfaced defined data type;

– any defined data types represented by a SELECT data type whose selection list contains the interfaced defined data type.

EXAMPLE 81 – Implicit interface to a defined data type via a SELECT data type.

```
SCHEMA s1;
  TYPE sel1 = SELECT (e1,t1);
  END_TYPE;

  TYPE t1 = INTEGER;
  END_TYPE;

  ENTITY e1;
  ...
  END_ENTITY;
END_SCHEMA;

SCHEMA s2;
REFERENCE FROM s1 (t1);
END_SCHEMA;
```

Schema `s2` contains an explicit reference to `t1`, and since `sel1` is represented by a SELECT which contains `t1`, `sel1` is implicitly referenced.

11.4.3 Entity data type interfaces

When an entity data type is interfaced, the following are implicitly interfaced:

– all entity data types which are supertypes of the interfaced entity data type;

NOTE – The subtypes of the interfaced entity data type, whether or not they appear in a SUPERTYPE OF expression, are not implicitly interfaced as a result of this interface.

– all rules referring to the interfaced entity data type and zero or more other entity data types, all of which are either explicitly or implicitly interfaced in the current schema;

– any constants, defined data types, entity data types or functions used in the declaration of attributes of the interfaced entity data type;

– any constants, defined data types, entity data types or functions used within the domain rules of the interfaced entity data type;

– any defined data types represented by a SELECT data type which specifies the interfaced entity data type in its selection list.

Subtype/supertype graphs may be pruned as a result of only following the `SUBTYPE OF` links when collecting the implicit interfaces of an interfaced entity data type. The algorithm used to

calculate the allowed instantiations of the resulting pruned subtype/supertype graph is given in annex C.

11.4.4 Function interfaces

When a function is interfaced, the following are implicitly interfaced:

- any defined data types or entity data types used in the declaration of parameters for the interfaced function;

- any defined data types or entity data types used in the declaration of the returned type for the interfaced function;

- any defined data types or entity data types used in the declaration of local variables within the interfaced function;

- any constants, functions or procedures used within the interfaced function.

11.4.5 Procedure interfaces

When a procedure is interfaced, the following are implicitly interfaced:

- any defined data types or entity data types used in the declaration of parameters for the interfaced procedure;

- any defined data types or entity data types used in the declaration of local variables within the interfaced procedure;

- any constants, functions or procedures used within the interfaced procedure.

11.4.6 Rule interfaces

When a rule is interfaced, the following are implicitly interfaced:

- any defined data types or entity data types used in the declaration of local variables within the interfaced rule;

- any constants, functions or procedures used within the interfaced rule.

12 Expression

Expressions are combinations of operators, operands and function calls which are evaluated to produce a value.

Syntax:

```
204 expression = simple_expression [ rel_op_extended simple_expression ] .
269 rel_op_extended = rel_op | IN | LIKE .
268 rel_op = '<' | '>' | '<=' | '>=' | '<>' | '=' | ':<>:' | ':=:' .
287 simple_expression = term { add_like_op term } .
303 term = factor { multiplication_like_op factor } .
205 factor = simple_factor [ '**' simple_factor ] .
288 simple_factor = aggregate_initializer | entity_constructor |
                   enumeration_reference | interval | query_expression |
                   ( [ unary_op ] ( '(' expression ')' | primary ) ) .
308 unary_op = '+' | '-' | NOT .
256 primary = literal | ( qualifiable_factor { qualifier } ) .
244 multiplication_like_op = '*' | '/' | DIV | MOD | AND | '||' .
158 add_like_op = '+' | '-' | OR | XOR .
```

Some operators require one operand and other operators require two operands. Operators which require only one operand shall precede that operand. Operators which require two operands shall be written between those operands. This clause defines the operators and specifies the data types of the operands which may be used with each operator.

There are seven classes of operators:

a) Arithmetic operators accept number operands and produce number results. The data type of the resulting value of an arithmetic operation depends upon the operator and the data types of the operands (see 12.1).

b) Relational operators accept various data types as operands and produce LOGICAL (TRUE, FALSE or UNKNOWN) results.

c) BINARY operators accept BINARY operands and produce BINARY results.

d) LOGICAL operators accept LOGICAL operands and produce LOGICAL results.

e) STRING operators accept STRING operands and produce STRING results.

f) Aggregate operators combine aggregate values with other aggregate values or with individual elements in various ways and produce aggregate results.

g) Component reference and index operators extract components from entity instances and aggregate values.

Evaluation of an expression is governed by the precedence of the operators which form part of the expression.

Expressions enclosed by parentheses are evaluated before being treated as a single operand.

Evaluation proceeds from left to right, with the highest precedence being evaluated first. Table 10 specifies the precedence rules for all of the operators of *EXPRESS*. Operators in the same row have the same precedence, and the rows are ordered by decreasing precedence.

An operand between two operators of different precedence is bound to the operator with the higher precedence. An operand between two operators of the same precedence is bound to the one on the left.

EXAMPLE 82 – −10**2 is evaluated as (−10)**2 resulting in the value 100. 10/20*30 is evaluated as (10/20) * 30 resulting in the value 15.0.

12.1 Arithmetic operators

Arithmetic operators which require one operand are identity (+) and negation (-). The operand shall be of numeric type (NUMBER, INTEGER or REAL). When the operator is +, the result is equal to the operand, when the operator is -, the result is the negation of the operand. When the operand is indeterminate (?) the result is indeterminate (?) irrespective of which operator is used.

The arithmetic operators which require two operands are addition (+), subtraction (-), multiplication (*), real division (/), exponentiation (**), integer division (DIV), and modulo (MOD). The operands shall be of numeric type (NUMBER, INTEGER or REAL).

The addition, subtraction, multiplication, division and exponentiation operators perform the mathematical operations of the same name. With the exception of division they produce an integer result if both operands are of data type INTEGER, a REAL result otherwise (subject to neither operand evaluating to indeterminate (?)). Real division (/) produces a real result (subject to neither operand evaluating to indeterminate (?)).

Modulo and integer division produce an integer result (subject to neither operand evaluating to indeterminate (?)); if either operand is of data type REAL, it is truncated to an INTEGER before the operation, i.e. any fractional part is lost. If a and b are integers, it is always true that (a DIV b)*b + (a MOD b) = a. The absolute value of a MOD b shall be less than the absolute value of b, and the sign of a MOD b shall be the same as the sign of b.

If any operand to an arithmetic operator is indeterminate (?), the result of the operation shall be indeterminate (?).

Real number rounding

Rounding, when necessary, shall be determined from the precision p (either stated explicitly for

Table 10 – Operator precedence

Precedence	Description	Operators
1	Component references	[ ] . \
2	Unary operators	+ - NOT
3	Exponentiation	**
4	Multiplication/Division	* / DIV MOD AND ||
5	Addition/Subtraction	- + OR XOR
6	Relational	= <> <= >= < > :=: :<>: IN LIKE

NOTE – || is the complex entity construction operator.

a REAL type or an implementation limit as specified in annex E) using the following algorithm:

a) convert the number representation to exponential format with all leading zeros removed;

b) set the digit pointer k to point at the digit p places to the right of the decimal point.

c) if the value of the real is positive, do the following:

– if the digit at k is in the range 5..9, add 1 to the digit at $k-1$, ignore the digit at k and all digits after. Goto step e;

– if the digit at k is in the range 0..4, ignore digit at k and all digits after. Goto step h.

d) if the value of the real is negative, do the following:

– if the digit at k is in the range 6..9, add 1 to the digit at $k-1$, ignore the digit at k and all digits after. Goto step e;

– if the digit at k is in the range 0..5, ignore digit at k and all digits after. Goto step h.

e) set the digit pointer k to $k-1$.

f) if the digit at k is in the range 0..9, goto step h.

g) if the digit at k has the value 10, add 1 to the digit at $k-1$, set the digit at k to 0. Goto step e.

h) the number is now rounded.

NOTE – The effect of this rounding mechanism is to round 0.5 to 1 and -0.5 to 0.

EXAMPLE 83 – This example shows the effect of defining the number of significant digits in the fraction part of a real number; i.e., its precision.

```
LOCAL
  distance   : REAL(6);
  x1, y1, z1 : REAL;
  x2, y2, z2 : REAL;
END_LOCAL;
...
x1 := 0.; y1 := 0.; z1 := 0.;
x2 := 10.; y2 := 11.; z2 := 12.;
...
distance := SQRT((x2-x1)**2 + (y2-y1)**2 + (z2-z1)**2);
```

`distance` is computed to a value of `1.9104973...e+1` but has an actual value of `1.91050e+1` since the specification calls for six digits of precision; thus, only six significant digits are retained.

12.2 Relational operators

The relational operators consist of value comparison operators, instance comparison operators, membership (IN) and string match (LIKE). The result of a relational expression is a LOGICAL value (TRUE, FALSE or UNKNOWN). If either operand evaluates to indeterminate (?) the expression evaluates to UNKNOWN.

12.2.1 Value comparison operators

The value-comparison operators are:

- equal (=);
- not equal (<>);
- greater than (>);
- less than (<);
- greater than or equal (>=);
- less than or equal (<=).

These operators may be applied to numeric, logical, string, binary and enumeration item operands. In addition, = and <> may be applied to values of aggregate and entity data types. The two operands of a value comparison operator shall be data type compatible. See 12.11.

For two given values, a and b, `a <> b` is equivalent to `NOT (a = b)` for all data types. If a and b are neither aggregation nor entity data types, in addition:

a) one of the following is TRUE: `a < b`, `a = b` or `a > b`

b) `a <= b` is equivalent to `(a < b) OR (a = b)`

c) `a >= b` is equivalent to `(a > b) OR (a = b)`

12.2.1.1 Numeric comparisons

The value comparison operators, when applied to numeric operands, shall correspond to the mathematical ordering of the real numbers.

NOTE – Precision specifications are not considered when comparing two real numbers.

EXAMPLE 84 – Given:

```
a : REAL(3) := 1.23
b : REAL(5) := 1.2300;
```

the expression `a = b` evaluates to TRUE.

12.2.1.2 Binary comparisons

To compare two binary values, compare the bits in the same position in each value, starting with the first (leftmost) pair of bits then the bits at the second position, etc., until an unequal pair is found or until all bit pairs have been examined. If an unequal pair is found, the binary value containing the 0 bit is less than the other binary value. No additional comparison is needed. If no unequal pair is found, the binary value which is shorter (using the BLENGTH function) is considered less than the other binary value. If both binary values are of the same length and all pairs are equal, the two binary values are equal.

12.2.1.3 Logical comparisons

Comparisons of two LOGICAL (or BOOLEAN) values shall observe the following ordering:

FALSE < UNKNOWN < TRUE.

12.2.1.4 String comparisons

To compare two string values, compare the characters in the same position in each value, starting with the first (leftmost) pair of characters then the pair at the second position, etc., until an unequal pair is found or until all character pairs have been examined. If an unequal pair is found, the string value containing the lesser character (as defined by the ISO 10646-1 octet values for the characters) is considered less than the other string value. No additional comparison is needed. If no unequal pair is found, the string value that is shorter (using the LENGTH function) is considered less than the other string value. If both string values are of the same length and all pairs are equal, the two string values are equal.

12.2.1.5 Enumeration item comparisons

Value comparison of enumeration items is based on their relative position in the declaration of the enumeration data type. See rule (a) in 8.4.1.

12.2.1.6 Aggregate value comparisons

The value comparison operators which are defined for aggregate values are equal (`=`) and not equal (`<>`). Two aggregate values can be compared only if their data types are compatible, see 12.11.

All aggregate comparisons shall check the number of elements in each of the operands: if `SIZEOF(a) <> SIZEOF(b)`, the aggregates are not equal. The aggregate comparisons compare the elements of the aggregate value using value comparisons. If any of the element comparisons evaluate to FALSE, the aggregate comparison evaluates to FALSE. If one or more of the element comparisons for a particular aggregate comparison evaluate to UNKNOWN and the remaining comparisons all evaluate to TRUE, the aggregate comparison evaluates to UNKNOWN. Otherwise the aggregate comparison evaluates to TRUE.

The definition of aggregate equality depends on the aggregate data types being compared.

– two arrays a and b are equal if and only if each element of a is value equal to the element of b at the same position, i.e., a[i] = b[i] (12.6.1);

– two lists a and b are equal if and only if each element of a is value equal to the element of b at the same position;

– two bags or sets a and b are equal if and only if each element VALUE_IN a occurs the same number of times VALUE_IN b and each element VALUE_IN b also occurs the same number of times VALUE_IN a.

NOTE – A bag may be value equal to a set even when the bag contains multiple occurances of the same instance, this is due to the test being based on the number of value equal elements.

12.2.1.7 Entity value comparisons

Two entity instances are equal by value comparison if their corresponding attributes are value equal. Since entity instances may have attributes which are represented by entity data types, it is possible for instances to be self referential, in which case the entity instances are equal by value comparison if all the attributes which are represented by simple data types have the same values and the same attributes in both entity instances are self referential.

More precisely, suppose two instances l and r are to be compared. If l :=: r, l = r. Otherwise, make the following definitions:

– Define an ordering on the population of instances under consideration. In practice this population will be finite, so an ordering can be constructed.

– For the purposes of this discussion, define an aggregate indexing operator which observes this ordering such that for any aggregate agg and for any indices i and j, the condition i < j is equivalent to the condition agg[i] < agg[j].

– Define a reference path to be a sequence of one or more attribute or index references. To apply a reference path s to an instance i, write s(i). s(i) is evaluable if no reference other than the last one produces indeterminate (?).

Then the value of l = r is determined by the first of the following conditions which holds:

a) If TYPEOF(l) <> TYPEOF(r), l = r is FALSE.

b) If there is a reference path s such that exactly one of s(l) and s(r) is evaluable, l = r is FALSE.

c) If there is a reference path s such that both s(l) and s(r) produce simple type values, and if s(l) <> s(r), l = r is FALSE.

d) If there is a reference path s such that NOT EXISTS(s(l)) or NOT EXISTS(s(r)), l = r is UNKNOWN.

e) Otherwise, l = r is TRUE.

EXAMPLE 85 – The algorithm outlined below is one possible implementation of the value comparison test described above. This algorithm is given for illustrative purposes and is not intended to prescribe any particular type of implementation.

Let `l` and `r` be variables of type GENERIC within this algorithm.

a) Initialise `l` to be the left hand entity instance and `r` be the right hand entity instance.

b) If the instances are the same instance, i.e., `l :=: r`, the expression evaluates to TRUE.

c) Initialise an empty list `plist` to contain ordered pairs of entity instance identifiers.

NOTE 1 – The representation of instance identifiers is implementation specific.

d) Compare `l` and `r` using the deep equal algorithm specified below.

e) The expression evaluates to the value returned by the deep equality algorithm.

Deep equality algorithm

a) If `l`, `r` or both are indeterminate (`?`) the algorithm returns UNKNOWN.

b) If `TYPEOF(l) <> TYPEOF(r)`, the algorithm returns FALSE.

c) If `l` and `r` are not entity instances, the algorithm returns `l = r`, using the appropriate equality test.

d) If `l` and `r` are the same entity instance, i.e., `l :=: r`, the algorithm returns TRUE.

e) If the pair of instances `(l, r)` appears in `plist`, the algorithm returns TRUE.

f) If the pair `(l, r)` does not appear in `plist`, do the following:

(1) Add the pair `(l, r)` to `plist`.

(2) For each of the attributes `a` defined for `l` and `r`, compare `l.a` and `r.a` using the deep equality algorithm, letting `l = l.a` and `r = r.a`.

NOTE 2 – This is the recursive call.

(3) If the deep equality algorithm for any of the attributes in step (f2) returns a FALSE result, the result of the current invocation of the algorithm is FALSE. Otherwise, if the algorithm returns UNKNOWN for any of the attributes, the result of this invocation is UNKNOWN. Otherwise, the result of this call is TRUE.

NOTE 3 – This ensures that if any of the comparisons are FALSE the result is FALSE. If all comparisons are TRUE , the result is TRUE. When any comparison is UNKNOWN and all other comparisons are TRUE the result is UNKNOWN.

EXAMPLE 86 – The local variables i1 and i2 are of type loop_of_integer and when assigned as in this example they are not value equal.

```
ENTITY loop_of_integer;
  int : INTEGER;
  next : loop_of_integer;
END_ENTITY;

...

LOCAL
  i1, i2 : loop_of_integer ;
END_LOCAL;

...

i1 := loop_of_integer(5,loop_of_integer(3,SELF));
i2 := loop_of_integer(3,loop_of_integer(5,SELF));

IF i1 = i2 THEN -- evaluates to false
```

Entity value comparison can be applied to entity instances and to group-qualified (see 12.7.4) entity instances. For entity instances, the attributes of all subtypes and supertypes of the instances under consideration shall be compared. For group-qualified entity instances, only those attributes which are declared as attributes in the entity declaration for the entity data type named in the group qualifier shall be compared (this does not include inherited attributes which are redeclared in the specified entity data type).

12.2.2 Instance comparison operators

The instance comparison operators are:

– instance equal (:=:);

– instance not equal (:<>:).

These operators may be applied to numeric, logical, string, binary, enumeration, aggregate and entity data type operands. The two operands of an instance comparison operator shall be data type compatible. See 12.11.

For two given operands, a and b, (a :<>: b) is equivalent to NOT (a :=: b) for all data types.

The instance comparison operators when applied to numeric, logical, string, binary and enumeration data types are equivalent to the corresponding value comparison operators. That is, (a :=: b) is equivalent to (a = b) and (a :<>: b) is equivalent to (a <> b) for these data types.

12.2.2.1 Aggregate instance comparison

The instance comparison operators which are defined for aggregate values are equal (`:=:`) and not equal (`:<>:`). Two aggregate values can be compared only if their data types are compatible, see 12.11.

All aggregate comparisons shall check the number of elements in each of the operands: if `SIZEOF(a) <> SIZEOF(b)`, the aggregates are not equal. The aggregate comparisons compare the elements of the aggregate value using instance comparisons. If any of the element comparisons evaluate to FALSE, the aggregate comparison evaluates to FALSE. If one or more of the element comparisons for a particular aggregate comparison evaluate to UNKNOWN and the remaining comparisons all evaluate to TRUE, the aggregate comparison evaluates to UNKNOWN. Otherwise the aggregate comparison evaluates to TRUE.

The definition of aggregate instance equality depends on the aggregate data types being compared.

– two arrays `a` and `b` are equal if and only if each element of `a` is the same instance as the element of `b` at the same position, i.e., `a[i] :=: b[i]` (12.6.1);

– two lists `a` and `b` are equal if and only if each element of `a` is the same instance as the element of `b` at the same position;

– two bags `a` and `b` are instance equal if and only if each element IN `a` occurs the same number of times IN `b` and each element IN `b` also occurs the same number of times IN `a`;

– two sets `a` and `b` are instance equal if and only if each element in `a` is IN `b` and each element in `b` is also IN `a`;

– A bag is instance equal to a set if and only if each element in the set occurs only once IN the bag, and the bag contains no elements which are not in the set.

EXAMPLE 87 – Instance comparison of two arrays.

```
LOCAL
  a1, a2 : ARRAY [1:10] OF b;
END_LOCAL;
...
IF (a1 :=: a2) THEN ...
```

12.2.2.2 Entity instance comparison

The entity instance equal (`:=:`) and entity instance not equal (`:<>:`) operators accept two compatible entity instances and evaluate to a LOGICAL value.

`a :=: b` evaluates to TRUE if `a` evaluates to the same entity instance as `b`, i.e., the implementation dependent identifiers are the same. It evaluates to FALSE if `a` evaluates to a different entity instance than `b`. It evaluates to UNKNOWN if either operand is indeterminate (`?`).

Unless otherwise noted, entity instance comparison shall be used when two entity instances are compared such as during aggregate comparisons and UNIQUE rule checking.

EXAMPLE 88 – All children have mothers but some children have either brothers or sisters. This could be modelled by

```
ENTITY child
SUBTYPE OF (person);
  mother : female; -- we are not interested in more than one generation
  father : male;
END_ENTITY;

ENTITY sibling
SUBTYPE OF (child);
  siblings : SET [1:?] sibling;
WHERE
              -- make sure that the current entityinstance
              -- is not one of its siblings

not_identical: SIZEOF ( QUERY ( i <* siblings | i :=: SELF ) ) = 0;
              -- make sure that each of the siblings shares either a mother
              -- or a father with the current entity instance
same_parent  : SIZEOF ( QUERY ( i <* siblings |
                              ( i.mother :=: SELF.mother ) OR
                              ( i.father :=: SELF.father ) ) =
               SIZEOF ( siblings ));
END_ENTITY;
```

12.2.3 Membership operator

The membership operator IN tests an item for membership in some aggregate and evaluates to a LOGICAL value. The right-hand operand shall evaluate to a value of an aggregation data type, and the left-hand operand shall be compatible with the base type of this aggregate value. `e IN agg` is evaluated as follows:

a) if either operand is indeterminate (`?`) the expression evaluates to UNKNOWN;

b) if there exists a member `agg[i]` such that `e :=: agg[i]`, the expression evaluates to TRUE;

c) if there exists a member `agg[i]` which is indeterminate (`?`) the expression evaluates to UNKNOWN;

d) otherwise the expression evaluates to FALSE.

NOTE – The function VALUE_IN (see 15.28) may be used to determine whether or not an element of an aggregation has a specific value.

Modeller-defined membership testing may be specified via a pair of functions, called for example `my_equal` (see the note in 8.2.5) and `my_in`, as shown in the following pseudo-code.

```
FUNCTION my_in(c:AGGREGATE OF GENERIC:gen; v:GENERIC:gen): LOGICAL;
  (* Returns UNKNOWN if v or c is indeterminate (?)
```

```
     Else returns TRUE if any element of c has the 'value' v
     Else returns UNKNOWN if any comparison is UNKNOWN
     Otherwise returns FALSE *)
LOCAL
  result   : LOGICAL;
  unknownp : BOOLEAN := FALSE;
END_LOCAL
IF ((NOT EXISTS(v)) OR (NOT EXISTS(c)) THEN
  RETURN(UNKNOWN); END_IF;
REPEAT i := LOINDEX(c) TO HIINDEX(c);
  result := my_equal(v, c[i]);
  IF (result = TRUE) THEN
    RETURN(result); END_IF;
  IF (result = UNKNOWN) THEN
    unknownp := TRUE; END_IF;
END_REPEAT;
IF (unknownp) THEN
  RETURN(UNKNOWN);
ELSE
  RETURN(FALSE);
END_IF;
END_FUNCTION;
```

This could, for example, be used as:

```
LOCAL
  v : a;
  c : SET OF a;
END_LOCAL;
...
IF my_in(c, v) THEN ...
```

12.2.4 Interval expressions

An interval expression tests whether or not a value falls within a given interval. It contains three operands, which shall be compatible (see 12.11). The operands shall be of a type which has a defined ordering, i.e., the simple types (see 8.1), and defined data types whose underlying types are either simple types or enumeration types.

Syntax:

```
229 interval = '{' interval_low interval_op interval_item interval_op
               interval_high '}' .
232 interval_low = simple_expression .
233 interval_op = '<' | '<=' .
231 interval_item = simple_expression .
230 interval_high = simple_expression .
```

NOTE – The interval expression

```
{ interval_low interval_op interval_item interval_op interval_high }
```

is semantically equivalent to

```
(interval_low interval_op interval_item) AND
(interval_item interval_op interval_high)
```

Assuming `interval_item` is evaluated only once in the second expression.

The interval expression evaluates to a LOGICAL which has the value TRUE if both relational operations evaluate to TRUE. It evaluates to FALSE if either relational operation evaluates to FALSE, and it evaluates to UNKNOWN if any operand is indeterminate (`?`).

EXAMPLE 89 – The following tests if the value of `b` is greater than 5.0 and less than or equal to 100.0.

```
LOCAL
  b : REAL := 20.0;
END_LOCAL;
...
IF { 5.0 < b <= 100.0 } THEN -- evaluates to TRUE
...
```

12.2.5 Like operator

The LIKE operator compares two string values using the pattern matching algorithm described below and evaluates to a LOGICAL value. The left operand is the target string. The right operand is the pattern string.

The pattern matching algorithm is defined as follows. Each character of the pattern string is compared to the corresponding character(s) of the target string. If any pair of corresponding characters does not match, the match fails and the expression evaluates to FALSE.

Certain special characters in the pattern string may match more than one character in the target string. These characters are defined in table 11. All corresponding characters must be identical or match as defined in table 11 for the expression to evaluate to TRUE. If either operand is indeterminate (`?`) the expression evaluates to UNKNOWN.

When any of the special pattern matching characters is itself to be matched, the pattern shall contain a pattern escape sequence. A pattern escape sequence shall consist of the escape character (`\`) followed by the special character to be matched.

EXAMPLE 90 – To match the character `@`, the escape sequence `\@` is used.

The following examples illustrate these pattern matching characters

EXAMPLES

91 – If `a := '\AAAA';` the following hold:

```
a LIKE '\\AAAA'   --> TRUE
a LIKE '\AAAA'    --> FALSE
a LIKE '\\A?AA'   --> TRUE
a LIKE '\\!\\AAA' --> TRUE
```

```
a LIKE '\\&'     --> TRUE
a LIKE '\$'      --> FALSE
```

92 – If `a := 'The quick red fox';`, the following holds:

```
a LIKE '$$$$'   --> TRUE
```

93 – If `a := 'Page 407';` the following holds:

```
a LIKE '$*'     --> TRUE
```

12.3 Binary operators

In addition to the relational operators, defined in 12.2.1.2, two further operations are defined for BINARY data types: indexing (`[ ]`) and concatenation (`+`) .

12.3.1 Binary indexing

The binary indexing operator takes two operands, the binary value being indexed and the index specification, and evaluates to a binary value of length `(index_2 - index_1 + 1)`. The resulting binary value is equivalent to the sequence bits at position `index_1` through `index_2` inclusive. If a binary value of length one is required, only `index_1` need be specified. An index of 1 indicates the leftmost bit.

Syntax:

```
226 index_qualifier = '[' index_1 [ ':' index_2 ] ']' .
224 index_1 = index .
223 index = numeric_expression .
225 index_2 = index .
```

Table 11 – Pattern matching characters

Character	Meaning
@	Matches any letter
^	Matches any upper case letter
?	Matches any character
&	Matches remainder of string
#	Matches any digit
$	Matches a substring terminated by a space character or end-of-string
*	Matches any number of characters
\	Begins a pattern escape sequence
!	Negation character (used with the other characters)

Rules and restrictions:

a) `index_1` shall evaluate to a positive integer value.

b) $1 \leq$ `index_1` $\leq$ `BLENGTH(` binary value `)`, otherwise indeterminate (`?`) is returned.

c) `index_2`, when specified, shall evaluate to positive integer value.

d) The `index_1` $\leq$ `index_2` $\leq$ `BLENGTH(` binary value `)`, otherwise indeterminate (`?`) is returned.

EXAMPLE 94 – The fourth bit of a binary called `image` could be examined by

```
image := %01010101

IF image[4]=%1 THEN ...-- evaluate to TRUE
IF image[4:4]=%1 THEN ... --equivalent expression
```

EXAMPLE 95 – The fourth through tenth bits of a binary called `image` could be examined by

```
IF image[4:10]=%1011110 THEN ...
```

12.3.2 Binary concatenation operator

The binary concatenation operator (`+`) is a binary operator which combines two binary values together. Both operands shall evaluate to a binary values, and the expression evaluates to a binary value containing the concatenation of the two operands with the first operand appearing on the left.

EXAMPLE 96 – Binary values may be concatenated as follows:

```
image := %101000101 + %101001 ;
(* image now contains the binary %101000101101001 *)
```

12.4 Logical operators

The logical operators consist of NOT, AND, OR and XOR. Each produces a logical result. The AND, OR and XOR operators require two logical operands, and the NOT operator requires one logical operand.

12.4.1 NOT operator

The NOT operator requires one logical operand (to the right of the NOT operator) and evaluates to the logical value as shown in table 12.

12.4.2 AND operator

The AND operator requires two logical operands and evaluates to a logical value as shown in table 13. The AND operator is commutative.

12.4.3 OR operator

The OR operator requires two logical operands and evaluates to a logical value as shown in table 14. The OR operator is commutative.

12.4.4 XOR operator

The XOR operator requires two logical operands and evaluates to a logical value as shown in table 15. The XOR operator is commutative.

12.5 String operators

In addition to the relational operators defined in 12.2.1.4 and 12.2.5, two further operations are defined for STRING types: indexing (`[ ]`) and concatenation (`+`) .

12.5.1 String indexing

The string indexing operator takes two operands, the string value being indexed and the index specification, and evaluates to a string value of length `(index_2 - index_1 + 1)`. The resulting string value is equivalent to the sequence of characters at position `index_1` through `index_2` inclusive. If a string value of length one is required, only `index_1` need be specified. An index of 1 indicates the leftmost character.

Syntax:

```
226 index_qualifier = '[' index_1 [ ':' index_2 ] ']' .
224 index_1 = index .
223 index = numeric_expression .
225 index_2 = index .
```

Rules and restrictions:

a) `index_1` shall evaluate to a positive integer value.

Table 12 – NOT operator

Operand Value	Result Value
TRUE	FALSE
UNKNOWN	UNKNOWN
FALSE	TRUE

b) $1 \leq$ index_1 $\leq$ LENGTH(string value), otherwise indeterminate (?) is returned.

c) index_2, when specified, shall evaluate to positive integer value.

d) The index_1 $\leq$ index_2 $\leq$ LENGTH(string value), otherwise indeterminate (?) is returned.

EXAMPLE 97 – The seventh character of a string called name could be examined by

```
IF name[7]="00125FE1" THEN ... -- assuming 10646 representation
IF name[7:7]="00125FE1" THEN ... -- equivalent expression
```

EXAMPLE 98 – The seventh through tenth characters of a string called name could be examined by

```
IF name[7:10]='Some' THEN ...
```

12.5.2 String concatenation operator

The string concatenation operator (+) is a string operator which combines two strings together. Both operands shall evaluate to a string values, and the expression evaluates to a string value containing the concatenation of the two operands with the first operand appearing on the left.

EXAMPLE 99 – String values may be concatenated as follows:

```
name := 'ABC' + ' ' + 'DEF' ;
(* name now contains the string 'ABC DEF' *)
```

12.6 Aggregate operators

The aggregate operators are indexing ([ ]), intersection (*), union (+), difference (-), subset (<=), superset (>=) and QUERY. These operators are defined in the following subclauses. The relational operators equal (=), not equal (<>), instance equal (:=:), instance not equal (:<>:) and IN, defined in 12.2, are also applicable to all aggregate values.

Table 13 – AND operator

Operand1 Value	Operand2 Value	Result Value
TRUE	TRUE	TRUE
TRUE	UNKNOWN	UNKNOWN
TRUE	FALSE	FALSE
UNKNOWN	TRUE	UNKNOWN
UNKNOWN	UNKNOWN	UNKNOWN
UNKNOWN	FALSE	FALSE
FALSE	TRUE	FALSE
FALSE	UNKNOWN	FALSE
FALSE	FALSE	FALSE

Table 14 – OR operator

Operand1 Value	Operand2 Value	Result Value
TRUE	TRUE	TRUE
TRUE	UNKNOWN	TRUE
TRUE	FALSE	TRUE
UNKNOWN	TRUE	TRUE
UNKNOWN	UNKNOWN	UNKNOWN
UNKNOWN	FALSE	UNKNOWN
FALSE	TRUE	TRUE
FALSE	UNKNOWN	UNKNOWN
FALSE	FALSE	FALSE

NOTE – Several of the aggregate operations require implicit comparisons of the elements contained within aggregate values; instance comparison is used in all such cases.

12.6.1 Aggregate indexing

The aggregate indexing operator takes two operands, the aggregate value being indexed and the index specification, and evaluates to a single element from the aggregate value. The data type of the element selected is the base type of the aggregate value being indexed.

Syntax:

```
226 index_qualifier = '[' index_1 [ ':' index_2 ] ']' .
224 index_1 = index .
223 index = numeric_expression .
225 index_2 = index .
```

Rules and restrictions:

a) `index_2` shall not be present; it is only possible to index a single element from an aggregate value.

Table 15 – XOR operator

Operand1 Value	Operand2 Value	Result Value
TRUE	TRUE	FALSE
TRUE	UNKNOWN	UNKNOWN
TRUE	FALSE	TRUE
UNKNOWN	TRUE	UNKNOWN
UNKNOWN	UNKNOWN	UNKNOWN
UNKNOWN	FALSE	UNKNOWN
FALSE	TRUE	TRUE
FALSE	UNKNOWN	UNKNOWN
FALSE	FALSE	FALSE

b) The index_1 shall evaluate to an integer value.

c) LOINDEX(aggregate value) ≤ index_1 ≤ HIINDEX(aggregate value), otherwise indeterminate (?) is returned.

d) If the type of the aggregate value is an ARRAY or a LIST, the expression evaluates to the element of the aggregate value at the position indicated by index_1.

e) If the type of aggregate value is a BAG or a SET, for each index_1 between LOINDEX(aggregate value) and HIINDEX(aggregate value), the expression shall evaluate to a different element of the aggregate value.

f) Repeated aggregate index on the same aggregate value with the same index_1 shall result in the same element being returned only if the aggregate value has not been modified. If the aggregate value has been modified for aggregation data types BAG or SET the results of repeated aggregate index on the modified aggregate value are unpredictable.

EXAMPLE 100 – Indexing for bags and sets may be used to iterate over all of the values in the aggregate value.

```
a_set : SET [1:20] OF INTEGER;
result : INTEGER;

result := 1;
REPEAT FOR index := LOINDEX(a_set) TO HIINDEX(a_set);
    result := result * a_set[index];
END_REPEAT;
```

On exit from the REPEAT statement, result holds the product of all of the integers in a_set.

12.6.2 Intersection operator

The intersection operator (*) accepts two aggregate value operands and evaluates to an aggregate value. The allowed operand types and corresponding result type are given in table 16. The resulting aggregate value is implicitly declared as an aggregate whose type is as specified in table 16 with bounds of [0..?]. The base data types of the operands shall be compatible (see 12.11). If the intersection of the two operands contains no elements, the size of the resulting aggregate value shall be zero.

If either of the operands is a set, the result shall be a set which contains each element which appears IN both of the operands.

If both the operands are bags and a particular element e occurs m times IN one bag and n times IN the other (where m is less or equal to n), the result shall contain m occurrences of e.

12.6.3 Union operator

The union operator (+) accepts two operands, one of which must be an aggregate value, and evaluates to an aggregate value. The allowed operand types and the corresponding result type

Table 16 – Intersection operator – operand and result types

First operand	Second operand	Result
Bag	Bag	Bag
Bag	Set	Set
Set	Set	Set
Set	Bag	Set

are given in table 17. The union operation is defined by the type of the operands and their order as follows:

– If one of the operands (`E`) is type compatible with the base type of the other operand (`A`), the operand `E` is added to `A` as follows:

- If `A` is a set value, the resulting set is `A`, with `E` added to the set only if `E` is not IN `A`.
- If `A` is a list value, the resulting list is `A`, with `E` inserted at position 1 if `E` was the left-hand operand and at position SIZEOF(`A`+1) if `E` was the right-hand operand.

  NOTE 1 – The resulting list may contain duplicate elements, even if the list operand was declared as `LIST OF UNIQUE`.

- If `A` is a bag value, the resulting bag is `A`, with `E` added.

– If both operands are compatible lists, the resulting list is the left-hand operand, with the right-hand operand appended to the end.

NOTE 2 – The resulting list may contain duplicate elements, even if the operands are both declared as `LIST OF UNIQUE`.

– If the left-hand operand is a set value, the resulting set is produced by initially setting the result to be the left-hand operand, for each element of the right-hand operand which is not IN the result adding that element to the result set.

– If the left-hand operand is a bag value, the result is the left-hand operand plus all the elements of the right-hand operand.

12.6.4 Difference operator

The difference operator (`-`) accepts two operands, the left-hand operand of which must be an aggregate value, and evaluates to an aggregate value. The allowed operand types and the corresponding result type are given in table 18. The resulting aggregate value contains the elements of the first operand except for those elements of the second operand, i.e., for each element of the second operand which is IN the first operand that element is removed from the first operand. The resulting aggregate value is implicitly declared as an aggregate whose type

Table 17 – Union operator – operand and result types

First operand	Second operand	Result
Bag	Bag	Bag
Bag	Element	Bag
Element	Bag	Bag
Bag	Set	Bag
Bag	List	Bag
Set	Set	Set
Set	Element	Set
Element	Set	Set
Set	Bag	Set
Set	List	Set
List	List	List[1]
Element	List	List[2]
List	Element	List[3]

NOTES

1 – First element of the second list follows the last element of the first list.

2 – New element becomes first in resultant list.

3 – New element becomes last in resultant list.

is as specified in table 18 with bounds of `[0..?]`. The base type of the operands shall be compatible (see 12.11). The data type of the returned aggregate value shall be the same as that of the first operand. If both the operands are bags and a particular element `e` occurs `m` times IN the first operand and `n` times IN the second operand, the result shall contain `m-n` occurrences of `e` if `m` is greater than `n`, and no occurences of `e` if `m` is less than or equal to `n`. If the second operand contains elements not in the first operand, these elements are ignored and do not form part of the resulting aggregate value.

Table 18 – Difference operator – operand and result types

First operand	Second operand	Result
Bag	Bag	Bag
Bag	Set	Bag
Bag	Element	Bag
Set	Set	Set
Set	Bag	Set
Set	Element	Set

EXAMPLE 101 – If A is a bag of integers `[1,2,1,3]`,

```
A - 1
```

evaluates to `[1,2,3]` which is equivalent to `[2,1,3]`.

12.6.5 Subset operator

The subset operator (`<=`) accepts two operands as defined in table 19 and evaluates to a LOGICAL. The expression evaluates to TRUE if and only if, for any element `e` which occurs `n` times IN the first operand, `e` occurs at least `n` times IN the second operand. The expression evaluates to UNKNOWN if either operand is indeterminate (`?`) and evaluates to FALSE otherwise.

The operands shall be of compatible types (see 12.11).

12.6.6 Superset operator

The superset operator (`>=`) accepts two operands as defined in table 19 and evaluates to a LOGICAL. The expression evaluates to TRUE if and only if, for any element `e` which occurs `n`

Table 19 – Subset and superset operators - operand types

First operand	Second operand
Bag	Bag
Bag	Set
Set	Bag
Set	Set

times IN the second operand, `e` occurs at least `n` times IN the first operand. The expression evaluates to UNKNOWN if either operand is indeterminate (`?`) and evaluates to FALSE otherwise.

The operands shall be of compatible types (see 12.11).

`b >= a` shall be exactly equivalent to `a <= b`.

12.6.7 Query expression

The QUERY expression applies a `logical_expression` individually against each element of an aggregate value, and evaluates to an aggregate value which contains the elements for which the `logical_expression` evaluates to TRUE. This has the effect of resulting in a subset of the original aggregate value where all of the elements of the subset satisfy the condition expressed by the logical expression.

Syntax:

```
264 query_expression = QUERY '(' variable_id '<*' aggregate_source '|'
                       logical_expression ')' .
160 aggregate_source = simple_expression .
241 logical_expression = expression .
```

Rules and restrictions:

a) The `variable_id` is implicitly declared as a variable within the scope of the query expression.

NOTE – This variable does not have to be declared elsewhere, and it does not persist outside the expression.

b) The `aggregate_source` shall evaluate to an aggregate value (ARRAY, BAG, LIST or SET).

c) The third operand (`logical_expression`) shall be an expression which evaluates to a LOGICAL result.

Elements are taken one by one from the source aggregate and replace the `variable_id` in the `logical_expression`. The `logical_expression` is then evaluated. If `logical_expression` evaluates to TRUE the element is added to the result; otherwise, it is not. This is repeated for every element of the source aggregate. The result aggregate value is populated according to the specific kind of aggregation data type:

Array: The result array has the same base type and bounds as the source array but the array elements are OPTIONAL. Each element is initially indeterminate (`?`). Any element in the source for which `logical_expression` evaluates to TRUE is then placed at the corresponding index position in the result.

Bag: The result bag has the same base type and upper bound as the source bag. The lower bound is zero. The result bag is initially empty. Any element in the source for which `logical_expression` evaluates to TRUE is then added to the result.

List: The result list has the same base type and upper bound as the source list. The lower bound is zero. The result list is initially empty. Any element in the source for which `logical_expression` evaluates to TRUE is then added to the end of the result. The order of the source list is preserved.

Set: The result set has the same base type and upper bound as the source set. The lower bound is zero. The result set is initially empty. Any element in the source for which `logical_expression` evaluates to TRUE is then added to the result.

EXAMPLE 102 – Assuming `colour` is a defined type which has as it's underlying type an ENUMERATION which includes `pink` and `scarlet`. The following could be used to extract from an array of `colours` those which are either `pink` or `scarlet`.

```
LOCAL
  colours : ARRAY OF colour;
  reds    : ARRAY OF OPTIONAL colour;
END_LOCAL;
...
reds := QUERY ( element <* colours | ( element = pink ) OR
                                     ( element = scarlet ) ) ;
...
```

EXAMPLE 103 – This rule uses a query expression to examine all instances of entity type point. The resulting set contains all instances of point located at the origin.

```
RULE two_points_at_origin FOR (point);
WHERE
  SIZEOF(QUERY(temp <* point | temp = point(0.0,0.0,0.0))) = 2;
END_RULE;
```

This example shows use of three implicit declarations. The first is the variable `point` which is implicitly declared as the set of all point instances by the rule header. The second is the variable `temp` which holds successive elements of the aggregate value `point` during evaluation of the query expression. The third is the constructor `point` resulting from its entity declaration.

12.7 References

When an item visible in the local scope is to be used locally, the item shall be referred to by the identifier declared for that item.

12.7.1 Simple references

A simple reference is simply the name (identifier) given to an item in the current scope.

The items which can be referred to in this manner are:

– Attributes within an entity declaration*;

– Constants*;

- Elements from an enumeration type*;
- Entities**;
- Functions*;
- Local variables within the body of an algorithm*;
- Parameters within the body of an algorithm*;
- Procedures;
- Rules;
- Schemas within an interface specification;
- Types.

Those items marked (*) may be referenced in this manner within an expression. Entities (marked **) may be referenced either as constructor (see 9.2.5), or as a local variable in a global rule (see 69).

EXAMPLE 104 – Valid simple references

```
line     (* entity type *)
Circle   (* entity type *)
RED      (* enumeration item *)
z_depth  (* attribute *)
```

12.7.2 Prefixed references

In the case where the same name for an enumeration item is declared in more than one defined data type visible in the same scope, (see clause 10), the enumeration item name shall be prefixed with the identifier of its defined data type in order to uniquely identify it. The prefixed reference is the defined data type name followed by a full stop (.), followed by the enumeration item name.

EXAMPLE 105 – This example shows how the enumeration item `red` is uniquely identified for use as `stop_signal`.

```
TYPE traffic_light = ENUMERATION OF (red, amber, green);
END_TYPE;

TYPE rainbow = ENUMERATION OF
                (red, orange, yellow, green, blue, indigo, violet);
END_TYPE;

stop_signal : traffic_light := traffic_light.red;

ink_colour : rainbow := blue;
```

12.7.3 Attribute references

The attribute reference (.) provides a reference to a single attribute within an entity instance. The expression to the left of the attribute reference shall evaluate to an entity instance or a partial complex entity value. The identifier of the attribute to be referenced is specified following the full stop (.).

Syntax:

```
169 attribute_qualifier = '.' attribute_ref .
```

Attribute referencing returns the value of the specified attribute within the entity instance or partial complex entity value when used within an expression. If the specified attribute is not present in the entity instance or partial complex entity value, indeterminate (?) is returned.

If used on the left-hand side of an assignment statement, or in a procedure as a VAR parameter, it provides a reference within an entity instance where a value is to be placed. Unpredictable results occur if the specified attribute is not present in the entity instance during an assignment statement.

EXAMPLE 106 – This example shows the use of attribute referencing.

```
ENTITY point;
  x, y, z, : REAL;
END_ENTITY;

ENTITY coloured_point
SUBTYPE OF (point);
  colour : colour;
END_ENTITY;
...
PROCEDURE foo;
LOCAL
  first : point := point(1.0,2.0,3.0)||coloured_point(red);
  second : coloured_point := point(1.0,2.0,3.0)||coloured_point(red);
  x_coord : REAL;
END_LOCAL;
...
x_coord := first.x; -- the value 1.0
IF first.colour = red THEN  -- colour is an invalid reference since it is
                            -- not an attribute of declared type of first
IF second.colour = red THEN -- TRUE since colour is a valid reference
```

12.7.4 Group references

The group reference (\) provides a reference to a partial complex entity value within a complex entity instance. The expression to the left of the group reference shall evaluate to a complex entity instance. The entity data type of the partial complex entity value to be referenced is specified following the reverse solidus (\).

Syntax:

```
219 group_qualifier = '\' entity_ref .
```

A group reference returns the partial complex entity value corresponding to the named entity data type within the complex entity instance being referenced when used within an expression. If the specified entity data type is not present in the complex entity instance being referenced, indeterminate (?) is returned. A group reference may be further qualified with an attribute reference. In this usage, the group reference specifies the scope of the attribute reference.

NOTE – This usage is required when a complex entity instance type has multiple attributes with the same name or when a select data type contains multiple entities with attributes having the same name.

Rules and restrictions:

A group reference which is not further qualified with an attribute reference shall appear as an operand of either the entity value comparison operator (=) or the complex entity instance constructor (||).

EXAMPLES

107 – This example shows how group referencing may be used for value comparison.

```
ENTITY E1
ABSTRACT SUPERTYPE;
  attrib1 : REAL;
  attrib2 : REAL;
  attrib3 : REAL;
END_ENTITY;

ENTITY E2
SUBTYPE OF (E1);
  attribA : INTEGER;
  attribB : INTEGER;
  attribC : INTEGER;
END_ENTITY;

LOCAL
  a : E1
  b : E2;
END_LOCAL;

  -- build complex instances of a and b
  -- using the complex entity instance construction operator

a := E1(0.0,1.0,2.0)||E2(1,2,3);
b := E1(0.0,1.0,2.0)||E2(3,2,1);

  -- test the values in a and b over
  -- those attributes declared in E1

a\E1 = b\E1 -- TRUE
```

```
(*
   equivalent to
        (a.attrib1 = b.attrib1) AND
        (a.attrib2 = b.attrib2) AND
        (a.attrib3 = b.attrib3)
*)
```

108 – This example shows how group referencing may be used to specify a particular entity data type to be looked in for an attribute name.

```
ENTITY foo1;
  attr : REAL;
END_ENTITY;

ENTITY foo2
 SUBTYPE OF (foo1);
 attr2 : BOOLEAN;
END_ENTITY;

TYPE t = BINARY (80) FIXED;
END_TYPE;

TYPE crazy=SELECT(foo2,t);
END_TYPE;

...

LOCAL
  v : crazy;
END_LOCAL;

...

IF 'THIS.FOO2' IN TYPEOF(v) THEN  -- this ensures that unpredictable results
                                  -- are not caused (sometimes called a guard).
   v\foo1.attr := 1.5;            -- assigns 1.5 to the attr attribute of v
  -- since attr is defined in foo1 the group
  -- reference has to use foo1.
END_IF;
```

12.8 Function call

The function call invokes a function. It consists of a function identifier possibly followed by an actual parameter list. The number, type and order of the actual parameters shall agree with the formal parameters defined for that function. A function call expression evaluates to the return value of the function when the actual parameters are substituted for the formal parameters in the function declaration.

A function invocation extends the instance space. Any instances created during the evaluation of the function shall be uniquely identifiable throughout the entire population of known

instances. Normally, an instance so created is not known outside of the creating invocation, and, in particular, is not a part of the instance population under consideration. The exception to this rule is when such an instance is returned as, or within, the result of the function call. In this case, the instance remains known at the point of invocation. If an instance is in this way returned to the schema level (i.e., as the value of a derived attribute or constant), the instance is considered as a part of the population under study.

Syntax:

```
207 function_call = ( built_in_function | function_ref ) [ actual_parameter_list ] .
157 actual_parameter_list = '(' parameter { ',' parameter } ')' .
251 parameter = expression .
```

Rules and restrictions:

The actual parameters passed shall be assignment compatible with the formal parameters.

EXAMPLE 109 – An example of function call usage.

```
ENTITY point;
  x, y, z : number;
END_ENTITY;

FUNCTION midpoint_of_line(l:line):point;
...
END_FUNCTION;

IF midpoint_of_line(L506).x = 9.0 THEN ...
                      -- using the attribute reference
                      -- operator directly on the result
END_IF;
```

12.9 Aggregate initializer

The aggregate initializer is used to establish a value for an array, bag, list or set. Square brackets enclose zero or more expressions which evaluate to values of a data type compatible with the base data type of the aggregate. When there are two or more values, commas shall separate them. A sparse array may be initialized using indeterminate (?) to represent the missing values. An aggregate initializer expression evaluates to an aggregate value containing the values specified as elements. The number of elements initialized shall agree with any bounds specified for the aggregation data type.

When the aggregate initializer is used which contains no elements, it is establishing an empty bag, list or set (this construct cannot be used to initialize empty arrays).

Syntax:

```
159 aggregate_initializer = '[' [ element { ',' element } ] ']' .
193 element = expression [ ':' repetition ] .
273 repetition = numeric_expression .
```

EXAMPLE 110 – Given the declaration

```
a : SET OF INTEGER;
```

a value may be assigned as:

```
a := [ 1, 3, 6, 9*8, -12 ] ; -- 9*8 is an expression = 72
```

When a number of consecutive values are the same, a repetition may be applied. This is represented by two expressions separated by the ':' character. The expression to the left of the colon is the value to be repeated. The expression to the right of the colon, the `repetition`, gives the number of times the left-hand value is to be repeated. This expression shall evaluate to non-negative integer value, and is evaluated once prior to initialization.

EXAMPLE 111 – Given the following declaration

```
a : BAG OF BOOLEAN ;
```

The two following statements are equivalent

```
a := [ TRUE:5 ];
a := [ TRUE, TRUE, TRUE, TRUE, TRUE ];
```

12.10 Complex entity instance construction operator

The complex entity instance construction operator (||) constructs an instance of a complex entity by combining the partial complex entity values. The partial complex entity values may be combined in any order. A complex entity instance construction operator expression evaluates to either a partial complex entity value or a complex entity instance. A partial complex entity data type may only occur once within one level of an entity instance construction operator expression. A partial complex entity value may occur at different levels if these are nested, i.e., if a partial complex entity value is being used to construct a complex entity instance which plays the role of an attribute within partial complex entity value being combined to form the complex entity instance. See annex B for further information on complex entity instances.

EXAMPLE 112 – Given:

```
ENTITY a
ABSTRACT SUPERTYPE;
  a1 : INTEGER;
END_ENTITY;

ENTITY b SUBTYPE OF (a);
  b1 : STRING;
END_ENTITY;

ENTITY c SUBTYPE OF (a);
  c1 : REAL;
END_ENTITY;
```

Then the following complex entity instances may be built.

```
LOCAL
  v1 : a ;
  v2 : c ;
END_LOCAL;

  v2 := a(2) || c(7.998e-5); -- this is of type a&c
  v1 := v2 || b('abc');      -- this is of type a&b&c
  v1 := v2\a || b("00002639"); -- this is of type a&b
  v1 := v1 || v2;          -- this is invalid since it would be of type a&b&a&c
```

12.11 Type compatibility

The operands of an operator shall be compatible with the data type(s) required by the operator. The data types of both operands of certain operators shall also be compatible with each other; these cases have been identified earlier in this clause. Data types may be compatible without being identical. Data types are compatible when one of the following conditions holds:

– the data types are the same;

– one data type is a subtype or specialization of the other (including defined data types which use a defined data type as the underlying data type);

– both data types are array data types with compatible base data types and identical bounds;

– both data types are list data types with compatible base data types.

– both data types are either bag or set data types with compatible base data types.

EXAMPLE 113 – Given the following definitions:

```
TYPE natural = REAL;
WHERE SELF >= 0.0;
END_TYPE;

TYPE positive = natural;
WHERE SELF > 0.0;
END_TYPE;

TYPE bag_of_natural = BAG OF natural;
END_TYPE;

TYPE set_of_up_to_five_positive = SET [0:5] OF positive;
END_TYPE;
```

the following are compatible data types:

Given	Is compatible with
REAL	INTEGER, REAL, NUMBER, natural, positive
natural	REAL, NUMBER, natural, positive
positive	REAL, NUMBER, natural, positive
bag_of_natural	BAG OF REAL, BAG OF NUMBER, BAG OF natural, BAG OF positive, SET OF REAL, SET OF NUMBER, SET OF natural, SET OF positive, bag_of_natural, set_of_up_to_five_positive
set_of_up_to_five_positive	BAG OF REAL, BAG OF NUMBER, BAG OF natural, BAG OF positive, SET OF REAL, SET OF NUMBER, SET OF natural, SET OF positive, bag_of_natural, set_of_up_to_five_positive

13 Executable statements

Executable statements define the actions of functions, procedures and rules. These statements act only on variables local to a FUNCTION, PROCEDURE or RULE. They are used to define the logic and actions required to support the definition of constraints, i.e., WHERE clauses and RULEs. These statements do not affect the entity instances within the domain as defined in clause 5. The executable statements are null, ALIAS, assignment, CASE, compound, ESCAPE, IF, procedure call, REPEAT, RETURN and SKIP.

Syntax:

```
291 stmt = alias_stmt | assignment_stmt | case_stmt | compound_stmt | escape_stmt |
           if_stmt | null_stmt | procedure_call_stmt | repeat_stmt | return_stmt |
           skip_stmt .
```

Executable statements can appear only within a FUNCTION PROCEDURE or RULE.

13.1 Null (statement)

An executable statement which consists solely of a semicolon is called a null statement. No action occurs as a result of a null statement.

Syntax:

```
247 null_stmt = ';' .
```

EXAMPLE 114 – The following is a potential use of the null statement.

```
IF a = 13 THEN
  ; -- this is a null statement.
ELSE
  b := 5 ;
END_IF ;
```

13.2 Alias statement

The ALIAS statement provides a local renaming capability for qualified variables and parameters.

Syntax:

```
164 alias_stmt = ALIAS variable_id FOR general_ref { qualifier } ';' stmt { stmt }
                 END_ALIAS ';' .
216 general_ref = parameter_ref | variable_ref .
```

Within the scope of an ALIAS statement, `variable_id` is implicitly declared to be an appropriately typed variable which holds the value referenced by the qualified identifier following the FOR keyword.

NOTE – The visibility rules for `variable_id` are described in 10.3.1.

EXAMPLE 115 – Assuming there is an entity data type `point` with attributes `x,y,z`, ALIAS may be used in the function `calculate_length` to reduce the length of the returned expression.

```
ENTITY line;
  start_point,
  end_point  : point;
END_ENTITY;

FUNCTION calculate_length (the_line : line) : real;
ALIAS s FOR the_line.start_point;
  ALIAS e FOR the_line.end_point;
    RETURN (SQRT((s.x - e.x)**2 + (s.y - e.y)**2 + (s.z - e.z)**2)) ;
  END_ALIAS;
END_ALIAS;
END_FUNCTION;
```

13.3 Assignment statement

The assignment statement is used to assign an instance to a local variable or parameter. The data type of the value assigned to the variable shall be assignment compatible with the variable or parameter.

NOTE – The assignment statement can be used to specify that two local variables are the same entity instance.

Syntax:

```
166 assignment_stmt = general_ref { qualifier } ':=' expression ';' .
216 general_ref = parameter_ref | variable_ref .
```

EXAMPLE 116 – The following are valid assignments.

```
LOCAL
  a, b : REAL ;
  p    : point;
END_LOCAL ;
```

```
...
a := 1.1 ;
b := 2.5 * a;
p.x := b ;
```

Assignment compatibility

The data type of the variable or parameter being assigned to and the resultant data type of an expression are said to be assignment compatible if any of the following hold true:

– the types are the same;

– the expression evaluates to a type which is a subtype or specialization of the type declared for the variable being assigned to;

– the declared type of the variable being assigned to is a defined type whose fundamental type is a select type, and the expression evaluates to a value of a type which is assignment compatible with one, or more, of the types specified in the select list of the select type.

The fundamental type of a defined type is the fundamental type of the underlying type, and the fundamental type of a type other than a defined type is the type itself.

– the variable is represented by a defined type whose fundamental type is a simple type and the expression evaluates to a value of this simple type.

– the variable is represented by an aggregation data type and the expression is an aggregate initializer the elements of which, if any, are assignment compatible with the base type of aggregation data type.

Those partial complex entity instances which are not valid complex entity instances (see annex B) cannot be assigned to parameters or variables, nor can they be passed as actual parameters to functions or procedures. This does not restrict the assignment of valid complex entity instances.

13.4 Case statement

The CASE statement is a mechanism for selectively executing statements based on the value of an expression. A statement is executed depending on the value of the `selector`. The case statement consists of an expression, which is the case selector, and a list of alternative actions, each one preceded by one or more expressions which are the case labels. The evaluated type of the case label shall be compatible with the type of the case selector. The first occurring statement having a case label that evaluates to the same value as the case selector is executed. At most, one case-action is executed. If none of the case labels evaluates to the same value as the case selector:

– if the OTHERWISE clause is present, the statement associated with OTHERWISE is executed;

– if the OTHERWISE clause is not present, no statement associated with the case action is executed.

Syntax:

```
182 case_stmt = CASE selector OF { case_action } [ OTHERWISE ':' stmt ]
                END_CASE ';' .
283 selector = expression .
180 case_action = case_label { ',' case_label } ':' stmt .
181 case_label = expression .
```

Rules and restrictions:

The type of the evaluated value of the case labels shall be compatible with the type of the evaluated value of the case selector.

EXAMPLE 117 – A simple case statement using integer case labels.

```
LOCAL
  a : INTEGER ;
  x : REAL ;
END_LOCAL ;
...
a := 3 ;
x := 34.97 ;
CASE a OF
  1    : x := SIN(x) ;
  2    : x := EXP(x) ;
  3    : x := SQRT(x) ;  -- This is executed!
  4, 5 : x := LOG(x) ;
OTHERWISE : x := 0.0 ;
END_CASE ;
```

13.5 Compound statement

The compound statement is a sequence of statements delimited by BEGIN and END. A compound statement acts as a single statement.

NOTE – A compound statement does not define a new scope.

Syntax:

```
183 compound_stmt = BEGIN stmt { stmt } END ';' .
```

EXAMPLE 118 – A simple compound statement.

```
BEGIN
  a = a+1 ;
  IF a > 100 THEN
    a := 0 ;
  END_IF;
END ;
```

13.6 Escape statement

An ESCAPE statement causes an immediate transfer to the statement following the REPEAT statement in which it appears.

NOTE – This is the only way an REPEAT which according to the repeat control would be in an infinite loop may be terminated.

Syntax:

```
202 escape_stmt = ESCAPE ';' .
```

Rules and restrictions:

An ESCAPE statement shall only occur within the scope of a REPEAT statement.

EXAMPLE 119 – The ESCAPE statement passes control the statement following the END_REPEAT if `a < 0`.

```
REPEAT UNTIL (a=1);
  ...
  IF (a < 0) THEN
    ESCAPE;
  END_IF;
  ...
END_REPEAT;
-- control transferred to here
```

13.7 If ... Then ... Else statement

The IF...THEN...ELSE statement allows the conditional execution of statements based on an expression of type LOGICAL. When the `logical_expression` evaluates to TRUE the statement following THEN is executed. When the `logical_expression` evaluates to FALSE or UNKNOWN the statement following ELSE is executed if the ELSE clause is present. If the `logical_expression` evaluates to FALSE or UNKNOWN and the ELSE clause is omitted, control is passed to the next statement.

Syntax:

```
220 if_stmt = IF logical_expression THEN stmt { stmt } [ ELSE stmt { stmt } ]
              END_IF ';' .
241 logical_expression = expression .
```

EXAMPLE 120 – A simple IF statement.

```
IF a < 10 THEN
  c := c + 1;
ELSE
  c := c - 1;
END_IF;
```

13.8 Procedure call statement

The procedure call statement invokes a procedure. The actual parameters provided with the call shall agree in number, order and type with the formal parameters defined for that procedure.

Syntax:

```
257 procedure_call_stmt = ( built_in_procedure | procedure_ref )
                          [ actual_parameter_list ] ';' .
157 actual_parameter_list = '(' parameter { ',' parameter } ')' .
251 parameter = expression .
```

Rules and restrictions:

The actual parameters passed shall be assignment-compatible with the formal parameters.

EXAMPLE 121 – A call of the built-in procedure INSERT.

```
INSERT (point_list, this_point, here );
```

13.9 Repeat statement

The REPEAT statement is used to conditionally repeat the execution of a sequence of statements. Whether the repetition is started or continued is determined by evaluating the control condition(s). The control conditions are:

– finite iteration;

– while a condition is TRUE;

– until a condition is TRUE.

Syntax:

```
272 repeat_stmt = REPEAT repeat_control ';' stmt { stmt } END_REPEAT ';' .
271 repeat_control = [ increment_control ] [ while_control ] [ until_control ] .
222 increment_control = variable_id ':=' bound_1 TO bound_2 [ BY increment ] .
174 bound_1 = numeric_expression .
175 bound_2 = numeric_expression .
221 increment = numeric_expression .
```

These controls can be used in combination to specify the conditions that terminate the repetition.

These conditions are evaluated as follows to control the iterations:

a) Upon entering the REPEAT statement, if the increment control is present, the increment control statement is evaluated as described in 13.9.1.

b) The WHILE control expression, if present, is evaluated. If the result is TRUE (or no WHILE control exists), the body of the REPEAT statement is executed. If the result is FALSE or UNKNOWN the execution of the REPEAT statement is terminated.

c) When the execution of the body of the REPEAT statement is complete, any UNTIL control expression is evaluated. If the resulting value is TRUE the iteration stops and the execution of the REPEAT statement is complete. If the result is FALSE or UNKNOWN the control of the REPEAT statement is returned to the increment control. If no UNTIL control is present, the control of the REPEAT statement is returned to the increment control.

d) If increment control is present, the value of the loop variable is incremented by `increment`. If the loop variable is between `bound_1` and `bound_2`, including the bounds themselves, control passes to (b) above, otherwise the execution of the REPEAT statement is terminated.

EXAMPLE 122 – This example shows how more than one controlling condition can be used in a REPEAT statement. The statement iterates until either the desired tolerance is achieved, or one hundred cycles, whichever occurs first; i.e., iteration stops when the solution does not converge fast enough.

```
REPEAT i:=1 TO 100 UNTIL epsilon < 1.E-6;
  ...
  epsilon := ...;
END_REPEAT;
```

13.9.1 Increment control

The increment control causes the body of the repeat statement to be executed for successive values in a sequence. Upon entry to the repeat statement, the implicitly declared variable `variable_id`, of type number, is set to the value of `bound_1`. After each iteration, the value of `variable_id` is set to `variable_id + increment`. If `increment` is not specified, the default value of one (1) is used. If `variable_id` is between `bound_1` and `bound_2` (including the case where `variable_id = bound_2`), the execution of the repeat statement proceeds.

Syntax:

```
222 increment_control = variable_id ':=' bound_1 TO bound_2 [ BY increment ] .
174 bound_1 = numeric_expression .
175 bound_2 = numeric_expression .
221 increment = numeric_expression .
```

Rules and restrictions:

a) The `numeric_expression`s representing the `bound_1`, `bound_2` and `increment` shall evaluate to numeric values.

b) The `numeric_expression`s representing the bounds and the increment are evaluated once on entry to the REPEAT statement.

c) If any of the `numeric_expression`s representing the bounds or the increment have the value indeterminate (`?`) the REPEAT statement is not executed.

d) Upon first evaluation of the increment control statement the following conditions are checked for:

– if `increment` is positive and `bound_1` > `bound_2`, the REPEAT statement is not executed;

– if `increment` is negative and `bound_1` < `bound_2`, the REPEAT statement is not executed;

– if `increment` is zero (0), the REPEAT statement is not executed;

– if none of the above are true, the REPEAT statement is executed until the value of `variable_id` is outside the bounds specified or one of the other REPEAT control statements, if present, terminates the execution.

e) The loop variable is initialized to `bound_1` at the beginning of the first iteration cycle and incremented by `increment` at the beginning of each subsequent cycle.

f) The body of the REPEAT statement shall not modify the value of the loop variable.

g) The REPEAT statement establishes a local scope in which the loop variable `variable_id` is implicitly declared as a number variable. Thus any declaration of `variable_id` in the surrounding scope is hidden within the REPEAT statement, and the value of the loop variable is not available outside the REPEAT statement.

13.9.2 While control

The WHILE control initiates and continues the execution of the body of the REPEAT statement while the control expression is TRUE. The expression is evaluated before each iteration.

If the WHILE control is present and the expression evaluates to FALSE or UNKNOWN the body is not executed.

Syntax:

```
316 while_control = WHILE logical_expression .
241 logical_expression = expression .
```

Rules and restrictions:

a) The `logical_expression` shall evaluate to a LOGICAL value.

b) The `logical_expression` is re-evaluated at the beginning of each iteration.

13.9.3 Until control

The UNTIL control continues the execution of the body of the REPEAT statement until the control expression evaluates to TRUE. The expression shall be evaluated after each iteration.

If the UNTIL control is the only control present, at least one iteration shall always be executed.

Syntax:

```
312 until_control = UNTIL logical_expression .
241 logical_expression = expression .
```

Rules and restrictions:

a) The `logical_expression` shall evaluate to a LOGICAL value.

b) The `logical_expression` is re-evaluated at the end of each iteration.

13.10 Return statement

The RETURN statement terminates the execution of a FUNCTION or PROCEDURE. A RETURN statement within a function shall specify an expression. The value produced by evaluating this expression is the result of the function and is returned to the point of invocation. The expression shall be assignment-compatible with the declared return type of the function. A RETURN statement within a procedure shall not specify an expression.

Syntax:

```
276 return_stmt = RETURN [ '(' expression ')' ] ';' .
```

Rules and restrictions:

The RETURN statement shall only occur in a PROCEDURE or FUNCTION.

EXAMPLE 123 – Assorted RETURN statements.

```
RETURN(50) ;          (* from a function *)
RETURN(work_point) ;  (* from a function *)
RETURN ;              (* from a procedure *)
```

13.11 Skip statement

A SKIP statement causes an immediate transfer to the end of the body of the REPEAT statement in which it appears. The control conditions are then evaluated as described in 13.9.

Syntax:

```
290 skip_stmt = SKIP ';' .
```

Rules and restrictions:

The SKIP statement shall only occur in the scope of the REPEAT statement.

EXAMPLE 124 – The SKIP statement passes control to the END_REPEAT statement which causes the UNTIL control to to be evaluated.

```
REPEAT UNTIL (a=1);
```

```
    ...
    IF (a < 0) THEN
      SKIP;
    END_IF;
    ... -- statements here are missed if a < 0
  END_REPEAT;
```

14 Built-in constants

EXPRESS provides several built-in constants, which are defined here.

NOTE – The built-in constants are considered to have exact values, even though such a value might not be representable on a computer.

14.1 Constant e

CONST_E is a REAL constant representing the mathematical value e, the base of the natural logarithm function (ln). Its value is given by the following mathematical formula:

$$e = \sum_{i=0}^{\infty} i!^{-1}$$

14.2 Indeterminate

The indeterminate symbol (?) stands for an ambiguous value. It is compatible with all data types.

NOTE – The most common use of indeterminate (?) is as the upper bound specification of a bag, list or set. This usage represents the notion that the size of the aggregate value defined by the aggregation data type is unbounded.

14.3 False

FALSE is a LOGICAL constant representing the logical notion of falsehood. It is compatible with the BOOLEAN and LOGICAL data types.

14.4 Pi

PI is a REAL constant representing the mathematical value π, the ratio of a circle's circumference to its diameter.

14.5 Self

SELF refers to the current entity instance or type value. SELF may appear within an entity declaration, a type declaration or an entity constructor.

NOTE – SELF is not a constant, but behaves as one in every context in which it can appear.

14.6 True

TRUE is a LOGICAL constant representing the logical notion of truth. It is compatible with the BOOLEAN and LOGICAL data types.

14.7 Unknown

UNKNOWN is a LOGICAL constant representing that there is insufficient information available to be able to evaluate a logical condition. It is compatible with the LOGICAL data type, but not with the BOOLEAN data type.

15 Built-in functions

All functions (and mathematical operations in general) are assumed to evaluate to exact results.

The prototype for each of the built-in functions is given to show the type of the formal parameters and the result.

15.1 Abs - arithmetic function

```
FUNCTION ABS ( V:NUMBER ) : NUMBER;
```

The ABS function returns the absolute value of a number.

Parameters : V is a number.

Result : The absolute value of V. The returned data type is identical to the data type of V.

EXAMPLE 125 – `ABS ( -10 )  --> 10`

15.2 ACos - arithmetic function

```
FUNCTION ACOS ( V:NUMBER ) : REAL;
```

The ACOS function returns the angle given a cosine value.

Parameters : V is a number which is the cosine of an angle.

Result : The angle in radians ($0 \leq$ result $\leq \pi$) whose cosine is V.

Conditions : $-1.0 \leq$ V ≤ 1.0

EXAMPLE 126 – `ACOS ( 0.3 )  --> 1.266103...`

15.3 ASin - arithmetic function

```
FUNCTION ASIN ( V:NUMBER ) : REAL;
```

The ASIN function returns the angle given a sine value.

Parameters : V is a number which is the sine of an angle.

Result : The angle in radians ($-\pi/2 \leq$ result $\leq \pi/2$) whose sine is V.

Conditions : $-1.0 \leq$ V ≤ 1.0

EXAMPLE 127 – `ASIN ( 0.3 )  --> 3.04692...e-1`

15.4 ATan - arithmetic function

```
FUNCTION ATAN ( V1:NUMBER; V2:NUMBER ) : REAL;
```

The ATAN function returns the angle given a tangent value of V, where V is given by the expression $V = V1/V2$.

Parameters :

a) V1 is a number.

b) V2 is a number.

Result : The angle in radians ($-\pi/2 \leq$ result $\leq \pi/2$) whose tangent is V. If V2 is zero, the result is $\pi/2$ or $-\pi/2$ depending on the sign of V1.

Conditions : Both V1 and V2 shall not be zero.

EXAMPLE 128 – `ATAN ( -5.5, 3.0 ) --> -1.071449...`

15.5 BLength - binary function

```
FUNCTION BLENGTH ( V:BINARY ) : INTEGER;
```

The BLENGTH function returns the number of bits in a binary.

Parameters : V is a binary value.

Result : The returned value is the actual number of bits in the binary value passed.

EXAMPLE 129 –
```
LOCAL
  n : NUMBER;
  x : BINARY := %01010010 ;
END_LOCAL;
  ...
  n := BLENGTH ( x ); -- n is assigned the value 8
```

15.6 Cos - arithmetic function

```
FUNCTION COS ( V:NUMBER ) : REAL;
```

The COS function returns the cosine of an angle.

Parameters : V is a number which is an angle in radians.

Result : The cosine of V (-1.0 ≤ result ≤ 1.0).

EXAMPLE 130 – `COS ( 0.5 )  --> 8.77582...E-1`

15.7 Exists - general function

```
FUNCTION EXISTS ( V:GENERIC ) : BOOLEAN;
```

The EXISTS function returns TRUE if a value exists for the input parameter, or FALSE if no value exists for it. The EXISTS function is useful for checking if values have been given to OPTIONAL attributes, or if variables have been initialized.

Parameters : V is an expression which results in any type.

Result : TRUE or FALSE depending on whether V has an actual or indeterminate (?) value.

EXAMPLE 131 – `IF EXISTS ( a )  THEN ...`

15.8 Exp - arithmetic function

```
FUNCTION EXP ( V:NUMBER ) : REAL;
```

The EXP function returns e (the base of the natural logarithm system) raised to the power V.

Parameters : V is a number.

Result : The value e^V.

EXAMPLE 132 – `EXP ( 10 )  --> 2.202646...E+4`

15.9 Format - general function

```
FUNCTION FORMAT(N:NUMBER; F:STRING):STRING;
```

The FORMAT returns a formatted string representation of a number.

Parameters :

a) N is a number (integer or real).

b) F is a string containing formatting commands.

Result : A string representation of N formatted according to F. Rounding is applied to the string representation if necessary.

The formatting string contains special characters to indicate the appearance of the result. The formatting string can be written in three ways:

a) The formatting string can give a symbolic description of the output representation.

b) The formatting string can give a picture description of the output representation.

c) When the formatting string is empty, a standard output representation is produced.

15.9.1 Symbolic representation

The general form of a symbolic format is `[sign]width[.decimals]type`.

– `sign` indicates how the sign of the number is represented. If `sign` is either not specified or specified as a minus (`-`), the first character returned is a minus for negative numbers and a space for positive numbers (including zero). If `sign` is specified as a plus (`+`), the first character returned is a minus for negative numbers, a plus for positive numbers and a space for zero.

– `width` gives the total number of characters in the returned string. This shall be an integer number greater than two. If the `width` is specified with a preceding zero, the returned string will contain preceding zeros, otherwise preceding zeros are suppressed. If the number to be formatted requires more characters than specified by `width`, a string with the number of characters required by the format is returned.

– `decimals` gives the number of digits that are returned in the string to the right of the decimal point. If specified, the decimals shall be a positive integer number. If `decimals` is not specified, the decimal point and following digits are not returned in the string.

– `type` is a letter indicating the form of the number being represented in the string.

- if `type` is `I`, the number shall be represented as an integer and;
  - `decimals` shall not be specified;
  - the `width` specification shall be at least two;
- if `type` is `F`, the number shall be represented by a fixed point real and;
  - the `decimals`, if specified, shall be at least one;
  - if the `decimals` is not specified, the default value of two shall be assumed;
  - the `width` specification shall be at least four;
- if `type` is `E`, the number shall be represented by an exponential real and;
  - `decimals` shall always be specified for a `type` E;
  - the `decimals` specification shall be at least one;
  - the `width` specification shall be at least seven (7);
  - a zero prefixed `width` will result in the first two characters of the mantissa being `0.`;

* the exponent part shall be a minimum of two characters with an explicit sign;

* the displayed 'E' shall be in uppercase.

NOTE – Table 20 shows how formatting affects the appearance of different values.

Table 20 – Example symbolic formatting effects

Number	Format	Display	Comment
10	+7I	' +10'	Zero suppression
10	+07I	'+000010'	Zeros not suppressed
10	10.3E	' 1.000E+01'	
123.456789	8.2F	' 123.46'	
123.456789	8.2E	'1.23E+02'	
123.456789	08.2E	'0.12E+02'	Preceding zero forced
9.876E123	8.2E	'9.88E+123'	Exponent part is 3 characters and width ignored
32.777	6I	' 33'	Rounded

15.9.2 Picture representation

In the picture format each picture character corresponds to a character in the returned string. The characters used are given in table 21.

Table 21 – Picture formatting characters

# (hash mark)	represents a digit
, (comma)	a separator
. (dot)	a separator
+ - (plus and minus)	represents the sign
( ) (parentheses)	represents negation

The separators '.' and ',' are used as follows:

– if the ',' appears in the format string before the '.', the ',' represents a grouping character and the '.' represents the decimal character;

– if the '.' appears in the format string before the ',', the '.' represents a grouping character and the ',' represents the decimal character;

– if only one separator appears in the format string, this represents the decimal character.

Any other character is displayed without change.

NOTE – Table 22 shows how formatting affects the appearance of different values.

Table 22 – Example picture formatting effects

Number	Format	Display	Comment
10	###	' 10'	
10	(###)	' 10 '	parentheses ignored
-10	(###)	'(10)'	
7123.456	###,###.##	' 7,123.46'	USA notation
7123.456	###.###,##	' 7.123,46'	European notation

15.9.3 Standard representation

The standard representation for an integer number is '7I'. The standard representation for a real number is '10E'. See the description of symbolic representations above.

15.10 HiBound - arithmetic function

```
FUNCTION HIBOUND ( V:AGGREGATE OF GENERIC ) : INTEGER;
```

The HIBOUND function returns the declared upper index of an ARRAY or the declared upper bound of a BAG, LIST or SET.

Parameters : V is an aggregate value.

Result :

a) When V is an ARRAY the returned value is the declared upper index.

b) When V is a BAG, LIST or SET the returned value is the declared upper bound; if there are no bounds declared or the upper bound is declared to be indeterminate (?) indeterminate (?) is returned.

EXAMPLE 133 – Usage of HIBOUND function on nested aggregate values.

```
LOCAL
  a : ARRAY[-3:19] OF SET[2:4] OF LIST[0:?] OF INTEGER;
  h1, h2, h3 : INTEGER;
END_LOCAL;
  ...
  a[-3][1][1] := 2;        -- places a value in the list
  ...
  h1 := HIBOUND(a);        -- =19 (upper bound of array)
  h2 := HIBOUND(a[-3]);    -- = 4 (upper bound of set)
  h3 := HIBOUND(a[-3][1]); -- = ? (upper bound of list (unbounded))
```

15.11 HiIndex - arithmetic function

```
FUNCTION HIINDEX ( V:AGGREGATE OF GENERIC ) : INTEGER;
```

The HIINDEX function returns the upper index of an ARRAY or the number of elements in a BAG, LIST or SET.

Parameters : V is an aggregate value.

Result :

a) When V is an ARRAY, the returned value is the declared upper index.

b) When V is a BAG, LIST or SET, the returned value is the actual number of elements in the aggregate value.

EXAMPLE 134 – Usage of HIINDEX function on nested aggregate values.

```
LOCAL
  a : ARRAY[-3:19] OF SET[2:4] OF LIST[0:?] OF INTEGER;
  h1, h2, h3 : INTEGER;
END_LOCAL;
  a[-3][1][1] := 2;        -- places a value in the list
  h1 := HIINDEX(a);        -- = 19 (upper bound of array)
  h2 := HIINDEX(a[-3]);    -- =  1 (size of set) -- this is invalid with respect
                                                 -- to the bounds on the SET
  h3 := HIINDEX(a[-3][1]); -- =  1 (size of list)
```

15.12 Length - string function

```
FUNCTION LENGTH ( V:STRING ) : INTEGER;
```

The LENGTH function returns the number of characters in a string.

Parameters : V is a string value.

Result : The returned value is the number of characters in the string and shall be greater than or equal to zero.

EXAMPLE 135 – Usage of the LENGTH function.

```
LOCAL
  n : NUMBER;
  x1 : STRING := 'abc';
  x2 : STRING := "000025FF000101B5;
END_LOCAL;
  ...
  n := LENGTH ( x1 ); -- n is assigned the value 3
  n := LENGTH ( x2 ); -- n is assigned the value 2
```

15.13 LoBound - arithmetic function

```
FUNCTION LOBOUND ( V:AGGREGATE OF GENERIC ) : INTEGER;
```

The LOBOUND function returns the declared lower index of an ARRAY, or the declared lower bound of a BAG, LIST or SET.

Parameters : V is an aggregate value.

Result :

a) When V is an ARRAY the returned value is the declared lower index.

b) When V is a BAG, LIST or SET the returned value is the declared lower bound; if no lower bound is declared, zero (0) is returned.

EXAMPLE 136 – Usage of LOBOUND function on nested aggregate values.

```
LOCAL
  a : ARRAY[-3:19] OF SET[2:4] OF LIST[0:?] OF INTEGER;
  h1, h2, h3 : INTEGER;
END_LOCAL;
  ...
  h1 := LOBOUND(a);        -- =-3 (lower index of array)
  h2 := LOBOUND(a[-3]);    -- = 2 (lower bound of set)
  h3 := LOBOUND(a[-3][1]); -- = 0 (lower bound of list)
```

15.14 Log - arithmetic function

```
FUNCTION LOG ( V:NUMBER ) : REAL;
```

The LOG function returns the natural logarithm of a number.

Parameters : V is a number.

Result : A real number which is the natural logarithm of V.

Conditions : V > 0.0

EXAMPLE 137 – `LOG ( 4.5 )  --> 1.504077...E0`

15.15 Log2 - arithmetic function

```
FUNCTION LOG2 ( V:NUMBER ) : REAL;
```

The LOG2 function returns the base two logarithm of a number.

Parameters : V is a number.

Result : A real number which is the base two logarithm of V.

Conditions : V > 0.0

EXAMPLE 138 – `LOG2 ( 8 )  --> 3.00...E0`

15.16 Log10 - arithmetic function

```
FUNCTION LOG10 ( V:NUMBER ) : REAL;
```

The LOG10 function returns the base ten logarithm of a number.

Parameters : V is a number.

Result : A real number which is the base ten logarithm of V.

Conditions : V > 0.0

EXAMPLE 139 – `LOG10 ( 10 )  --> 1.00...E0`

15.17 LoIndex - arithmetic function

```
FUNCTION LOINDEX ( V:AGGREGATE OF GENERIC ) : INTEGER;
```

The LOINDEX function returns the lower index of an aggregate value.

Parameters : V is an aggregate value.

Result :

a) When V is an ARRAY the returned value is the declared lower index.

b) When V is a BAG, LIST or SET, the returned value is 1 (one).

EXAMPLE 140 – Usage of LOINDEX function on nested aggregate values.

```
LOCAL
  a : ARRAY[-3:19] OF SET[2:4] OF LIST[0:?] OF INTEGER;
  h1, h2, h3 : INTEGER;
END_LOCAL;
  ...
  h1 := LOINDEX(a);        -- =-3 (lower bound of array)
  h2 := LOINDEX(a[-3]);    -- = 1 (for set)
  h3 := LOINDEX(a[-3][1]); -- = 1 (for list)
```

15.18 NVL - null value function

```
FUNCTION NVL(V:GENERIC:GEN1; SUBSTITUTE:GENERIC:GEN1):GENERIC:GEN1;
```

The NVL function returns either the input value or an alternate value in the case where the input has a indeterminate (?) value.

Parameters :

a) V is an expression which is of any type.

b) SUBSTITUTE is an expression which shall not evaluate to indeterminate (?).

Result : When V is not indeterminate (?) that value is returned. Otherwise, SUBSTITUTE is returned.

EXAMPLE 141 – ENTITY unit_vector;

```
  x, y : REAL;
  z : OPTIONAL REAL;
WHERE
  x**2 + y**2 + NVL(z, 0.0)**2 = 1.0;
END_ENTITY;
```

The NVL function is used to supply zero (0.0) as the value of Z when Z is indeterminate (?).

15.19 Odd - arithmetic function

```
FUNCTION ODD ( V:INTEGER ) : LOGICAL;
```

The ODD function returns TRUE or FALSE depending on whether a number is odd or even.

Parameters : V is an integer number.

Result : When V MOD 2 = 1 TRUE is returned; otherwise FALSE is returned.

Conditions : Zero is not odd.

EXAMPLE 142 – ODD (121) --> TRUE

15.20 RolesOf - general function

```
FUNCTION ROLESOF ( V:GENERIC ) : SET OF STRING;
```

The ROLESOF function returns a set of strings containing the fully qualified names of the roles played by the specified entity instance. A fully qualified name is defined to be the name of the attribute qualified by the name of the schema and entity in which this attribute is declared (i.e. 'SCHEMA.ENTITY.ATTRIBUTE').

Parameters : V is any instance of an entity data type.

Result : A set of string values (in upper case) containing the fully qualified names of the attributes of the entity instances which use the instance V.

When a named data type is USE'd or REFERENCE'd, the schema and the name in that schema, if renamed, are also returned. Since USE statements may be chained, all the chained schema names and the name in each schema are returned.

EXAMPLE 143 – This example shows that a point might be used as the centre of a circle. The ROLESOF function determines what roles an entity instance actually plays.

```
SCHEMA that_schema;

ENTITY point;
  x, y, z : REAL;
END_ENTITY;

ENTITY line;
  start,
  end : point;
END_ENTITY;

END_SCHEMA;

SCHEMA this_schema;
USE FROM that_schema (point,line);

CONSTANT
  origin : point := point(0.0, 0.0, 0.0);
END_CONSTANT;

ENTITY circle;
  centre : point;
  axis   : vector;
  radius : REAL;
```

```
END_ENTITY;

  ...
LOCAL
  p : point := point(1.0, 0.0, 0.0);
  c : circle := circle(p, vector(1,1,1), 1.0);
  l : line := line(p, origin);
END_LOCAL;
  ...
IF 'THIS_SCHEMA.CIRCLE.CENTRE' IN ROLESOF(p) THEN   -- true
  ...
IF 'THIS_SCHEMA.LINE.START' IN ROLESOF(p) THEN      -- true
  ...
IF 'THAT_SCHEMA.LINE.START' IN ROLESOF(p) THEN      -- true
  ...
IF 'THIS_SCHEMA.LINE.END' IN ROLESOF(p) THEN      -- false
```

15.21 Sin - arithmetic function

```
FUNCTION SIN ( V:NUMBER ) : REAL;
```

The SIN function returns the sine of an angle.

Parameters : V is a number representing an angle expressed in radians.

Result : The sine of V (-1.0 $\leq$ result $\leq$ 1.0).

EXAMPLE 144 – `SIN ( PI )  --> 0.0`

15.22 SizeOf - aggregate function

```
FUNCTION SIZEOF ( V:AGGREGATE OF GENERIC ) : INTEGER;
```

The SIZEOF function returns the number of elements in an aggregate value.

Parameters : V is an aggregate value.

Result :

a) When V is an ARRAY the returned value is its declared number of elements in the aggregation data type.

b) When V is a BAG, LIST or SET, the returned value is the actual number of elements in the aggregate value.

EXAMPLE 145 –

```
LOCAL
  n : NUMBER;
  y : ARRAY[2:5] OF b;
END_LOCAL;
  ...
  n := SIZEOF (y); -- n is assigned the value 4
```

15.23 Sqrt - arithmetic function

```
FUNCTION SQRT ( V:NUMBER ) : REAL;
```

The SQRT function returns the non-negative square root of a number.

Parameters : V is any non-negative number.

Result : The non-negative square root of V.

Conditions : V ≥ 0.0

EXAMPLE 146 – `SQRT ( 121 )  --> 11.0`

15.24 Tan - arithmetic function

```
FUNCTION TAN ( V:NUMBER ) : REAL;
```

The TAN function returns the tangent of of an angle.

Parameters : V is a number representing an angle expressed in radians.

Result : The tangent of the angle. If the angle is $n\pi/2$, where n is an odd integer, indeterminate (?) is returned.

EXAMPLE 147 – `TAN ( 0.0 )  --> 0.0`

15.25 TypeOf - general function

```
FUNCTION TYPEOF ( V:GENERIC ) : SET OF STRING;
```

The TYPEOF function returns a set of strings that contains the names of all the data types of which the parameter is a member. Except for the simple data types (BINARY, BOOLEAN, INTEGER, LOGICAL, NUMBER, REAL, and STRING) and the aggregation data types (ARRAY, BAG, LIST, SET) these names are qualified by the name of the schema which contains the definition of the type.

NOTE 1 – The primary purpose of this function is to check whether a given value (variable, attribute value) can be used for a certain purpose, e.g. to ensure assignment compatibility between two values. It may also be used if different subtypes or specializations of a given type have to be treated differently in some context.

Parameters : V is a value of any type.

Result : The contents of the returned set of string values are the names (in upper case) of all types the value V is a member of. Such names are qualified by the name of the schema which contains the definition of the type (`'SCHEMA.TYPE'`) if it is neither a simple data type nor an aggregation data type. It may be derived by the following algorithm (which is given here for specification purposes rather than to prescribe any particular type of implementation):

a) The result set is initialized using the type name V belongs to (by declaration), including the schema name when the type name is a named data type. If V is a formal parameter, it is replaced by the corresponding actual parameter first. If V is a aggregate value, the type name is just the name of the aggregation data type (ARRAY, BAG, LIST, SET).

b) Repeat until the list stops growing:

(1) Repeat for all names in the result set:

– if the current name is the name of a simple data type: skip;

– if the current name is the name of an aggregation data type (ARRAY, BAG, LIST, SET): skip;

– if the current name is the name of an enumeration data type: skip;

– if the current name is the name of a select data type: the names of all the types (with the schema name) in the select list which are actually instantiated by `V` are added to the result set. (This may be more than one, as the select list may contain names of types which are compatible subtypes of a common supertype, or specialization of a common generalization.);

– if the current name is the name of any other sort of defined data type: the name of the type referenced by this type definition including the schema name where necessary is added to the result set. If the referenced type is an aggregation data type, this is just the name of the aggregation data type;

– if the current name is the name of an entity: the names of all those subtypes (including the schema name where necessary) actually instantiated by `V`, if any, are added to the result set.

(2) Repeat for all names in the result set:

– if the current name is the name of a subtype: the names of all its supertypes are added to the result set.

– if the current name is the name of a specialization: the names of all its generalizations are added to the result set.

(3) Repeat for all names in the result set:

– if the current name appears in the type list of at least one select data type: the names of all select data types, the definitions of which are based upon the type with the current name, are added to the result set.

(4) Repeat for all names in the result set:

– if the current name has been brought into its schema by means of REFERENCE or USE: the name in the interfaced schema qualified by the name of the interfaced schema is added to the result set. Since USE statements may be chained the name in all chained schemas qualified by the schema names are also added to the set.

c) Return result set.

If V evaluates to indeterminate (?) TYPEOF returns an empty set.

NOTE 2 – This function ends its work when it reaches an aggregation data type. It does not provide the information concerning the base type of the aggregate value. If needed, this information can be collected by applying TYPEOF to legal elements of the aggregate value.

EXAMPLE 148 – In the context of the following schema

```
SCHEMA this_schema;

  TYPE
    mylist = LIST [1 : 20] OF REAL;
  END_TYPE;
  ...
  LOCAL
    lst : mylist;
  END_LOCAL;
  ...

END_SCHEMA;
```

the following conditions are TRUE:

```
TYPEOF (lst)      = ['THIS_SCHEMA.MYLIST', 'LIST']
TYPEOF (lst [17]) = ['REAL', 'NUMBER']
```

EXAMPLE 149 – The effects of USE or REFERENCE are shown based on the previous example

```
SCHEMA another_schema;
  REFERENCE FROM this_schema (mylist AS hislist);
  ...
    lst : hislist;
  ...
END_SCHEMA;
```

Now we can say:

```
TYPEOF (lst) = ['ANOTHER_SCHEMA.HISLIST', 'THIS_SCHEMA.MYLIST', 'LIST']
```

15.26 UsedIn - general function

```
FUNCTION USEDIN ( T:GENERIC; R:STRING) : BAG OF GENERIC;
```

The USEDIN function returns each entity instance that uses a specified entity instance in a specified role.

Parameters :

a) T is any instance of any entity data type.

b) R is a string that contains a fully qualified attribute (role) name as defined in 15.20.

Result : Every entity instance that uses the specified instance in the specified role is returned in a bag.

An empty bag is returned if the instance T plays no roles or if the role R is not found.

When R is an empty string, every usage of T is reported. All relationships directed toward T are examined. When the relationship originates from an attribute with the name R, the entity instance containing that attribute is added to the result bag. Note that if T is not used, an empty bag is returned.

EXAMPLE 150 – This example shows how a rule might be used to test that there must be a point used as the centre of a circle at the origin. Note that this example uses the query expression (see 12.6.7) as the parameter to the SIZEOF function.

```
ENTITY point;
  x, y, z : REAL;
END_ENTITY;

ENTITY circle;
  centre : point;
  axis   : vector;
  radius : REAL;
END_ENTITY; ...

(*  This rule finds every point that is used as a circle centre, and then it
ensures that at least one of the points lies at the origin  *)

...

RULE example FOR (point);
LOCAL
 centre_points : SET OF circle := [];  -- an empty set of circle
END_LOCAL;
REPEAT i := LOINDEX(point) TO HIINDEX(point);
  centre_points := centre_points +
                   USEDIN(point[i],'THIS_SCHEMA.CIRCLE.CENTRE');
END_REPEAT;
WHERE R1 : SIZEOF(
             QUERY(                                          -- start query
               at_zero <* centre_points |                    -- get all points
               (at_zero.centre = point(0.0, 0.0, 0.0))       -- at 0,0,0
             )
          ) >= 1;                                          -- at least one
END_RULE;
```

15.27 Value - arithmetic function

```
FUNCTION VALUE ( V:STRING ) : NUMBER;
```

The VALUE function returns the numeric representation of a string.

Parameters : V is a string containing either a real or integer literal (see 7.5).

Result : A number corresponding to the string representation. If it is not possible to interpret the string as either a real or integer literal, indeterminate (?) is returned.

```
EXAMPLE 151 –     VALUE ( '1.234' ) --> 1.234 (REAL)
    VALUE ( '20' )    --> 20  (INTEGER)
    VALUE ( 'abc' )   --> ?   null
```

15.28 Value_in - membership function

```
FUNCTION VALUE_IN ( C:AGGREGATE OF GENERIC:GEN; V:GENERIC:GEN ) : LOGICAL;
```

The VALUE_IN function returns a LOGICAL value depending on whether or not a particular value is a member of an aggregation.

Parameters :

a) C is an aggregation of any type.

b) V is an expression which is assignment compatable with the base type of C.

Result :

a) If either V or C is indeterminate (?), UNKNOWN is returned.

b) If any element of C has a value equal to the value of V, TRUE is returned.

c) If any element of C is indeterminate (?), UNKNOWN is returned.

d) Otherwise FALSE is returned.

EXAMPLE 152 – The following test ensures that there is at least one point which is positioned at the origin.

```
LOCAL
  points : SET OF point;
END_LOCAL;
...
IF VALUE_IN(points, point(0.0, 0.0, 0.0)) THEN ...
```

15.29 Value_unique - uniqueness function

```
FUNCTION VALUE_UNIQUE ( V:AGGREGATE OF GENERIC) : LOGICAL;
```

The VALUE_UNIQUE function returns a LOGICAL value depending on whether or not the elements of an aggregation are value unique.

Parameters : V is an aggregation of any type.

Result :

a) If V is indeterminate (?), UNKNOWN is returned.

b) If any any two elements of V are value equal, FALSE is returned.

c) If any element of V is indeterminate (?), UNKNOWN is returned.

d) Otherwise TRUE is returned.

EXAMPLE 153 – The following test ensures tht each point is a set is at a different position, (by definition they are distinct, i.e., instance unique).

```
IF VALUE_UNIQUE(points) THEN ...
```

16 Built-in procedures

EXPRESS provides two built-in procedures, both of which are used to manipulate lists. The procedures are described in this clause.

The procedure head for each is given to show the data types of the formal parameters.

16.1 Insert

```
PROCEDURE INSERT ( VAR L:LIST OF GENERIC:GEN; E:GENERIC:GEN; P:INTEGER );
```

The INSERT procedure inserts an element at a particular position in a list.

Parameters :

a) L is the list value into which an element is to be inserted.

b) E is the instance to be inserted into L. E shall be compatible with the base type of L, as indicated by the type labels in the procedure head.

c) P is an integer giving the position in L at which E is to be inserted.

Result : L is modified by inserting E into L at the specified position. The insertion follows the existing element at position P, so when P = 0, E becomes the first element.

Conditions : 0 ≤ P ≤ SIZEOF(L)

16.2 Remove

```
PROCEDURE REMOVE ( VAR L:LIST OF GENERIC; P:INTEGER );
```

The REMOVE procedure removes an element from a particular position in a list.

Parameters :

a) L is the list value from which an element is to be removed.

b) P is an integer giving the position of the element in L to be removed.

Result : L is modified by removing the element found at the specified position P.

Conditions : 1 ≤ P ≤ SIZEOF(L)

Annex A
(normative)
EXPRESS language syntax

This annex defines the lexical elements of the language and the grammar rules which these elements shall obey.

NOTE – This syntax definition will result in ambiguous parsers if used directly. It has been written to convey information regarding the use of identifiers. The interpreted identifiers define tokens which are references to declared identifiers, and therefore should not resolve to simple_id. This requires a parser developer to provide a lookup table, or similar, to enable identifier reference resolution and return the required reference token to a grammar rule checker. This approach has been used to aid the implementors of parsers in that there should be no ambiguity with respect to the use of identifiers.

A.1 Tokens

The following rules specify the tokens used in *EXPRESS*. Except where explicitly stated in the syntax rules, no white space or remarks shall appear within the text matched by a single syntax rule in the following clauses: A.1.1, A.1.2, A.1.3 and A.1.5.

A.1.1 Keywords

This subclause gives the rules used to represent the keywords of *EXPRESS*.

NOTE – This subclause follows the typographical convention stated in 6.1 that each keyword is represented by a syntax rule whose left-hand side is that keyword in uppercase. Since string literals in the syntax rules are case-insensitive, these keywords may be given in a formal specification in upper, lower or mixed case.

```
 0 ABS = 'abs' .
 1 ABSTRACT = 'abstract' .
 2 ACOS = 'acos' .
 3 AGGREGATE = 'aggregate' .
 4 ALIAS = 'alias' .
 5 AND = 'and' .
 6 ANDOR = 'andor' .
 7 ARRAY = 'array' .
 8 AS = 'as' .
 9 ASIN = 'asin' .

10 ATAN = 'atan' .
11 BAG = 'bag' .
12 BEGIN = 'begin' .
13 BINARY = 'binary' .
14 BLENGTH = 'blength' .
15 BOOLEAN = 'boolean' .
16 BY = 'by' .
17 CASE = 'case' .
18 CONSTANT = 'constant' .
```

```
19 CONST_E = 'const_e' .

20 CONTEXT = 'context' .
21 COS = 'cos' .
22 DERIVE = 'derive' .
23 DIV = 'div' .
24 ELSE = 'else' .
25 END = 'end' .
26 END_ALIAS = 'end_alias' .
27 END_CASE = 'end_case' .
28 END_CONSTANT = 'end_constant' .
29 END_CONTEXT = 'end_context' .

30 END_ENTITY = 'end_entity' .
31 END_FUNCTION = 'end_function' .
32 END_IF = 'end_if' .
33 END_LOCAL = 'end_local' .
34 END_MODEL = 'end_model' .
35 END_PROCEDURE = 'end_procedure' .
36 END_REPEAT = 'end_repeat' .
37 END_RULE = 'end_rule' .
38 END_SCHEMA = 'end_schema' .
39 END_TYPE = 'end_type' .

40 ENTITY = 'entity' .
41 ENUMERATION = 'enumeration' .
42 ESCAPE = 'escape' .
43 EXISTS = 'exists' .
44 EXP = 'exp' .
45 FALSE = 'false' .
46 FIXED = 'fixed' .
47 FOR = 'for' .
48 FORMAT = 'format' .
49 FROM = 'from' .

50 FUNCTION = 'function' .
51 GENERIC = 'generic' .
52 HIBOUND = 'hibound' .
53 HIINDEX = 'hiindex' .
54 IF = 'if' .
55 IN = 'in' .
56 INSERT = 'insert' .
57 INTEGER = 'integer' .
58 INVERSE = 'inverse' .
59 LENGTH = 'length' .

60 LIKE = 'like' .
61 LIST = 'list' .
62 LOBOUND = 'lobound' .
63 LOCAL = 'local' .
64 LOG = 'log' .
65 LOG10 = 'log10' .
```

```
 66 LOG2 = 'log2' .
 67 LOGICAL = 'logical' .
 68 LOINDEX = 'loindex' .
 69 MOD = 'mod' .

 70 MODEL = 'model' .
 71 NOT = 'not' .
 72 NUMBER = 'number' .
 73 NVL = 'nvl' .
 74 ODD = 'odd' .
 75 OF = 'of' .
 76 ONEOF = 'oneof' .
 77 OPTIONAL = 'optional' .
 78 OR = 'or' .
 79 OTHERWISE = 'otherwise' .

 80 PI = 'pi' .
 81 PROCEDURE = 'procedure' .
 82 QUERY = 'query' .
 83 REAL = 'real' .
 84 REFERENCE = 'reference' .
 85 REMOVE = 'remove' .
 86 REPEAT = 'repeat' .
 87 RETURN = 'return' .
 88 ROLESOF = 'rolesof' .
 89 RULE = 'rule' .

 90 SCHEMA = 'schema' .
 91 SELECT = 'select' .
 92 SELF = 'self' .
 93 SET = 'set' .
 94 SIN = 'sin' .
 95 SIZEOF = 'sizeof' .
 96 SKIP = 'skip' .
 97 SQRT = 'sqrt' .
 98 STRING = 'string' .
 99 SUBTYPE = 'subtype' .

100 SUPERTYPE = 'supertype' .
101 TAN = 'tan' .
102 THEN = 'then' .
103 TO = 'to' .
104 TRUE = 'true' .
105 TYPE = 'type' .
106 TYPEOF = 'typeof' .
107 UNIQUE = 'unique' .
108 UNKNOWN = 'unknown' .
109 UNTIL = 'until' .

110 USE = 'use' .
111 USEDIN = 'usedin' .
112 VALUE = 'value' .
```

```
113 VALUE_IN = 'value_in' .
114 VALUE_UNIQUE = 'value_unique' .
115 VAR = 'var' .
116 WHERE = 'where' .
117 WHILE = 'while' .
118 XOR = 'xor' .
```

A.1.2 Character classes

The following rules define various classes of characters which are used in constructing the tokens in A.1.3.

```
119 bit = '0' | '1' .

120 digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' .
121 digits = digit { digit } .
122 encoded_character = octet octet octet octet .
123 hex_digit = digit | 'a' | 'b' | 'c' | 'd' | 'e' | 'f' .
124 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' .
125 lparen_not_star = '(' not_star .
126 not_lparen_star = not_paren_star | ')' .
127 not_paren_star = letter | digit | not_paren_star_special .
128 not_paren_star_quote_special = '!' | '"' | '#' | '$' | '%' | '&' | '+' | ',' |
                                 '-' | '.' | '/' | ':' | ';' | '<' | '=' | '>' |
                                 '?' | '@' | '[' | '\' | ']' | '^' | '_' | '`' |
                                 '{' | '|' | '}' | '~' .
129 not_paren_star_special = not_paren_star_quote_special | '''' .

130 not_quote = not_paren_star_quote_special | letter | digit | '(' | ')' | '*' .
131 not_rparen = not_paren_star | '*' | '(' .
132 not_star = not_paren_star | '(' | ')' .
133 octet = hex_digit hex_digit .
134 special = not_paren_star_quote_special | '(' | ')' | '*' | '''' .
135 star_not_rparen = '*' not_rparen .
```

A.1.3 Lexical elements

The following rules specify how certain combinations of characters are interpreted as lexical elements within the language.

```
136 binary_literal = '%' bit { bit } .
137 encoded_string_literal = '"' encoded_character { encoded_character } '"' .
138 integer_literal = digits .
139 real_literal = digits '.' [ digits ] [ 'e' [ sign ] digits ] .

140 simple_id = letter { letter | digit | '_' } .
141 simple_string_literal = \q { ( \q \q ) | not_quote | \s | \o } \q .
```

A.1.4 Remarks

The following rules specify the syntax of remarks in *EXPRESS*.

```
142 embedded_remark = '(*' { not_lparen_star | lparen_not_star | star_not_rparen |
                      embedded_remark } '*)' .
143 remark = embedded_remark | tail_remark .
144 tail_remark = '--' { \a | \s | \o } \n .
```

A.1.5 Interpreted identifiers

The following rules represent identifiers which are known to have a particular meaning (i.e., to be declared elsewhere as types or functions, etc.).

NOTE – It is expected that identifiers matching these syntax rules are known to an implementation. How the implementation obtains this information is of no concern to the definition of the language. One method of gaining this information is multipass parsing: the first pass collects the identifiers from their declarations, so that subsequent passes are then able to distinguish a `veriable_ref` from a `function_ref`, for example.

```
145 attribute_ref = attribute_id .
146 constant_ref = constant_id .
147 entity_ref = entity_id .
148 enumeration_ref = enumeration_id .
149 function_ref = function_id .

150 parameter_ref = parameter_id .
151 procedure_ref = procedure_id .
152 schema_ref = schema_id .
153 type_label_ref = type_label_id .
154 type_ref = type_id .
155 variable_ref = variable_id .
```

A.2 Grammar rules

The following rules specify how the previous lexical elements may be combined into constructs of *EXPRESS*. White space and/or remark(s) may appear between any two tokens in these rules. The primary syntax rule for *EXPRESS* is `syntax`.

```
156 abstract_supertype_declaration = ABSTRACT SUPERTYPE [ subtype_constraint ] .
157 actual_parameter_list = '(' parameter { ',' parameter } ')' .
158 add_like_op = '+' | '-' | OR | XOR .
159 aggregate_initializer = '[' [ element { ',' element } ] ']' .
160 aggregate_source = simple_expression .
161 aggregate_type = AGGREGATE [ ':' type_label ] OF parameter_type .
162 aggregation_types = array_type | bag_type | list_type | set_type .
163 algorithm_head = { declaration } [ constant_decl ] [ local_decl ] .
164 alias_stmt = ALIAS variable_id FOR general_ref { qualifier } ';' stmt { stmt }
                 END_ALIAS ';' .
165 array_type = ARRAY bound_spec OF [ OPTIONAL ] [ UNIQUE ] base_type .
166 assignment_stmt = general_ref { qualifier } ':=' expression ';' .
167 attribute_decl = attribute_id | qualified_attribute .
```

```
168 attribute_id = simple_id .
169 attribute_qualifier = '.' attribute_ref .

170 bag_type = BAG [ bound_spec ] OF base_type .
171 base_type = aggregation_types | simple_types | named_types .
172 binary_type = BINARY [ width_spec ] .
173 boolean_type = BOOLEAN .
174 bound_1 = numeric_expression .
175 bound_2 = numeric_expression .
176 bound_spec = '[' bound_1 ':' bound_2 ']' .
177 built_in_constant = CONST_E | PI | SELF | '?' .
178 built_in_function = ABS | ACOS | ASIN | ATAN | BLENGTH | COS | EXISTS | EXP |
                      FORMAT | HIBOUND | HIINDEX | LENGTH | LOBOUND | LOINDEX |
                      LOG | LOG2 | LOG10 | NVL | ODD | ROLESOF | SIN | SIZEOF |
                      SQRT | TAN | TYPEOF | USEDIN | VALUE | VALUE_IN |
                      VALUE_UNIQUE .
179 built_in_procedure = INSERT | REMOVE .

180 case_action = case_label { ',' case_label } ':' stmt .
181 case_label = expression .
182 case_stmt = CASE selector OF { case_action } [ OTHERWISE ':' stmt ]
              END_CASE ';' .
183 compound_stmt = BEGIN stmt { stmt } END ';' .
184 constant_body = constant_id ':' base_type ':=' expression ';' .
185 constant_decl = CONSTANT constant_body { constant_body } END_CONSTANT ';' .
186 constant_factor = built_in_constant | constant_ref .
187 constant_id = simple_id .
188 constructed_types = enumeration_type | select_type .
189 declaration = entity_decl | function_decl | procedure_decl | type_decl .

190 derived_attr = attribute_decl ':' base_type ':=' expression ';' .
191 derive_clause = DERIVE derived_attr { derived_attr } .
192 domain_rule = [ label ':' ] logical_expression .
193 element = expression [ ':' repetition ] .
194 entity_body = { explicit_attr } [ derive_clause ] [ inverse_clause ]
                [ unique_clause ] [ where_clause ] .
195 entity_constructor = entity_ref '(' [ expression { ',' expression } ] ')' .
196 entity_decl = entity_head entity_body END_ENTITY ';' .
197 entity_head = ENTITY entity_id [ subsuper ] ';' .
198 entity_id = simple_id .
199 enumeration_id = simple_id .

200 enumeration_reference = [ type_ref '.' ] enumeration_ref .
201 enumeration_type = ENUMERATION OF '(' enumeration_id { ',' enumeration_id } ')' .
202 escape_stmt = ESCAPE ';' .
203 explicit_attr = attribute_decl { ',' attribute_decl } ':' [ OPTIONAL ]
                  base_type ';' .
204 expression = simple_expression [ rel_op_extended simple_expression ] .
205 factor = simple_factor [ '**' simple_factor ] .
206 formal_parameter = parameter_id { ',' parameter_id } ':' parameter_type .
207 function_call = ( built_in_function | function_ref ) [ actual_parameter_list ] .
208 function_decl = function_head [ algorithm_head ] stmt { stmt } END_FUNCTION ';' .
```

```
209 function_head = FUNCTION function_id [ '(' formal_parameter
                   { ';' formal_parameter } ')' ] ':' parameter_type ';' .

210 function_id = simple_id .
211 generalized_types = aggregate_type | general_aggregation_types | generic_type .
212 general_aggregation_types = general_array_type | general_bag_type |
                                general_list_type | general_set_type .
213 general_array_type = ARRAY [ bound_spec ] OF [ OPTIONAL ] [ UNIQUE ]
                        parameter_type .
214 general_bag_type = BAG [ bound_spec ] OF parameter_type .
215 general_list_type = LIST [ bound_spec ] OF [ UNIQUE ] parameter_type .
216 general_ref = parameter_ref | variable_ref .
217 general_set_type = SET [ bound_spec ] OF parameter_type .
218 generic_type = GENERIC [ ':' type_label ] .
219 group_qualifier = '\' entity_ref .

220 if_stmt = IF logical_expression THEN stmt { stmt } [ ELSE stmt { stmt } ]
            END_IF ';' .
221 increment = numeric_expression .
222 increment_control = variable_id ':=' bound_1 TO bound_2 [ BY increment ] .
223 index = numeric_expression .
224 index_1 = index .
225 index_2 = index .
226 index_qualifier = '[' index_1 [ ':' index_2 ] ']' .
227 integer_type = INTEGER .
228 interface_specification = reference_clause | use_clause .
229 interval = '{' interval_low interval_op interval_item interval_op
              interval_high '}' .

230 interval_high = simple_expression .
231 interval_item = simple_expression .
232 interval_low = simple_expression .
233 interval_op = '<' | '<=' .
234 inverse_attr = attribute_decl ':' [ ( SET | BAG ) [ bound_spec ] OF ] entity_ref
                  FOR attribute_ref ';' .
235 inverse_clause = INVERSE inverse_attr { inverse_attr } .
236 label = simple_id .
237 list_type = LIST [ bound_spec ] OF [ UNIQUE ] base_type .
238 literal = binary_literal | integer_literal | logical_literal | real_literal |
             string_literal .
239 local_decl = LOCAL local_variable { local_variable } END_LOCAL ';' .

240 local_variable = variable_id { ',' variable_id } ':' parameter_type
                    [ ':=' expression ] ';' .
241 logical_expression = expression .
242 logical_literal = FALSE | TRUE | UNKNOWN .
243 logical_type = LOGICAL .
244 multiplication_like_op = '*' | '/' | DIV | MOD | AND | '||' .
245 named_types = entity_ref | type_ref .
246 named_type_or_rename = named_types [ AS ( entity_id | type_id ) ] .
247 null_stmt = ';' .
248 number_type = NUMBER .
```

```
249 numeric_expression = simple_expression .

250 one_of = ONEOF '(' supertype_expression { ',' supertype_expression } ')' .
251 parameter = expression .
252 parameter_id = simple_id .
253 parameter_type = generalized_types | named_types | simple_types .
254 population = entity_ref .
255 precision_spec = numeric_expression .
256 primary = literal | ( qualifiable_factor { qualifier } ) .
257 procedure_call_stmt = ( built_in_procedure | procedure_ref )
                        [ actual_parameter_list ] ';' .
258 procedure_decl = procedure_head [ algorithm_head ] { stmt } END_PROCEDURE ';' .
259 procedure_head = PROCEDURE procedure_id [ '(' [ VAR ] formal_parameter
                   { ';' [ VAR ] formal_parameter } ')' ] ';' .

260 procedure_id = simple_id .
261 qualifiable_factor = attribute_ref | constant_factor | function_call |
                       general_ref | population .
262 qualified_attribute = SELF group_qualifier attribute_qualifier .
263 qualifier = attribute_qualifier | group_qualifier | index_qualifier .
264 query_expression = QUERY '(' variable_id '<*' aggregate_source '|'
                     logical_expression ')' .
265 real_type = REAL [ '(' precision_spec ')' ] .
266 referenced_attribute = attribute_ref | qualified_attribute .
267 reference_clause = REFERENCE FROM schema_ref [ '(' resource_or_rename
                     { ',' resource_or_rename } ')' ] ';' .
268 rel_op = '<' | '>' | '<=' | '>=' | '<>' | '=' | ':<>:' | ':=:' .
269 rel_op_extended = rel_op | IN | LIKE .

270 rename_id = constant_id | entity_id | function_id | procedure_id | type_id .
271 repeat_control = [ increment_control ] [ while_control ] [ until_control ] .
272 repeat_stmt = REPEAT repeat_control ';' stmt { stmt } END_REPEAT ';' .
273 repetition = numeric_expression .
274 resource_or_rename = resource_ref [ AS rename_id ] .
275 resource_ref = constant_ref | entity_ref | function_ref | procedure_ref |
                 type_ref .
276 return_stmt = RETURN [ '(' expression ')' ] ';' .
277 rule_decl = rule_head [ algorithm_head ] { stmt } where_clause END_RULE ';' .
278 rule_head = RULE rule_id FOR '(' entity_ref { ',' entity_ref } ')' ';' .
279 rule_id = simple_id .

280 schema_body = { interface_specification } [ constant_decl ]
                { declaration | rule_decl } .
281 schema_decl = SCHEMA schema_id ';' schema_body END_SCHEMA ';' .
282 schema_id = simple_id .
283 selector = expression .
284 select_type = SELECT '(' named_types { ',' named_types } ')' .
285 set_type = SET [ bound_spec ] OF base_type .
286 sign = '+' | '-' .
287 simple_expression = term { add_like_op term } .
288 simple_factor = aggregate_initializer | entity_constructor |
                  enumeration_reference | interval | query_expression |
```

```
                    ( [ unary_op ] ( '(' expression ')' | primary ) ) .
289 simple_types = binary_type | boolean_type | integer_type | logical_type |
                   number_type | real_type | string_type .

290 skip_stmt = SKIP ';' .
291 stmt = alias_stmt | assignment_stmt | case_stmt | compound_stmt | escape_stmt |
           if_stmt | null_stmt | procedure_call_stmt | repeat_stmt | return_stmt |
           skip_stmt .
292 string_literal = simple_string_literal | encoded_string_literal .
293 string_type = STRING [ width_spec ] .
294 subsuper = [ supertype_constraint ] [ subtype_declaration ] .
295 subtype_constraint = OF '(' supertype_expression ')' .
296 subtype_declaration = SUBTYPE OF '(' entity_ref { ',' entity_ref } ')' .
297 supertype_constraint = abstract_supertype_declaration | supertype_rule .
298 supertype_expression = supertype_factor { ANDOR supertype_factor } .
299 supertype_factor = supertype_term { AND supertype_term } .

300 supertype_rule = SUPERTYPE subtype_constraint .
301 supertype_term = entity_ref | one_of | '(' supertype_expression ')' .
302 syntax = schema_decl { schema_decl } .
303 term = factor { multiplication_like_op factor } .
304 type_decl = TYPE type_id '=' underlying_type ';' [ where_clause ] END_TYPE ';' .
305 type_id = simple_id .
306 type_label = type_label_id | type_label_ref .
307 type_label_id = simple_id .
308 unary_op = '+' | '-' | NOT .
309 underlying_type = constructed_types | aggregation_types | simple_types |
                     type_ref .

310 unique_clause = UNIQUE unique_rule ';' { unique_rule ';' } .
311 unique_rule = [ label ':' ] referenced_attribute { ',' referenced_attribute } .
312 until_control = UNTIL logical_expression .
313 use_clause = USE FROM schema_ref [ '(' named_type_or_rename
                { ',' named_type_or_rename } ')' ] ';' .
314 variable_id = simple_id .
315 where_clause = WHERE domain_rule ';' { domain_rule ';' } .
316 while_control = WHILE logical_expression .
317 width = numeric_expression .
318 width_spec = '(' width ')' [ FIXED ] .
```

A.3 Cross reference listing

The production on the left is used in the productions indicated on the right.

```
  0 ABS                      | 178
  1 ABSTRACT                 | 156
  2 ACOS                     | 178
  3 AGGREGATE                | 161
  4 ALIAS                    | 164
  5 AND                      | 244 299
  6 ANDOR                    | 298
  7 ARRAY                    | 165 213
```

```
  8 AS                      | 246 274
  9 ASIN                    | 178

 10 ATAN                    | 178
 11 BAG                     | 170 214 234
 12 BEGIN                   | 183
 13 BINARY                  | 172
 14 BLENGTH                 | 178
 15 BOOLEAN                 | 173
 16 BY                      | 222
 17 CASE                    | 182
 18 CONSTANT                | 185
 19 CONST_E                 | 177

 20 CONTEXT                 |
 21 COS                     | 178
 22 DERIVE                  | 191
 23 DIV                     | 244
 24 ELSE                    | 220
 25 END                     | 183
 26 END_ALIAS               | 164
 27 END_CASE                | 182
 28 END_CONSTANT            | 185
 29 END_CONTEXT             |

 30 END_ENTITY              | 196
 31 END_FUNCTION            | 208
 32 END_IF                  | 220
 33 END_LOCAL               | 239
 34 END_MODEL               |
 35 END_PROCEDURE           | 258
 36 END_REPEAT              | 272
 37 END_RULE                | 277
 38 END_SCHEMA              | 281
 39 END_TYPE                | 304

 40 ENTITY                  | 197
 41 ENUMERATION             | 201
 42 ESCAPE                  | 202
 43 EXISTS                  | 178
 44 EXP                     | 178
 45 FALSE                   | 242
 46 FIXED                   | 318
 47 FOR                     | 164 234 278
 48 FORMAT                  | 178
 49 FROM                    | 267 313

 50 FUNCTION                | 209
 51 GENERIC                 | 218
 52 HIBOUND                 | 178
 53 HIINDEX                 | 178
 54 IF                      | 220
```

```
 55 IN                    | 269
 56 INSERT                | 179
 57 INTEGER               | 227
 58 INVERSE               | 235
 59 LENGTH                | 178

 60 LIKE                  | 269
 61 LIST                  | 215 237
 62 LOBOUND               | 178
 63 LOCAL                 | 239
 64 LOG                   | 178
 65 LOG10                 | 178
 66 LOG2                  | 178
 67 LOGICAL               | 243
 68 LOINDEX               | 178
 69 MOD                   | 244

 70 MODEL                 |
 71 NOT                   | 308
 72 NUMBER                | 248
 73 NVL                   | 178
 74 ODD                   | 178
 75 OF                    | 161 165 170 182 201 213 214 215 217 234 237 285 295
                            296
 76 ONEOF                 | 250
 77 OPTIONAL              | 165 203 213
 78 OR                    | 158
 79 OTHERWISE             | 182

 80 PI                    | 177
 81 PROCEDURE             | 259
 82 QUERY                 | 264
 83 REAL                  | 265
 84 REFERENCE             | 267
 85 REMOVE                | 179
 86 REPEAT                | 272
 87 RETURN                | 276
 88 ROLESOF               | 178
 89 RULE                  | 278

 90 SCHEMA                | 281
 91 SELECT                | 284
 92 SELF                  | 177 262
 93 SET                   | 217 234 285
 94 SIN                   | 178
 95 SIZEOF                | 178
 96 SKIP                  | 290
 97 SQRT                  | 178
 98 STRING                | 293
 99 SUBTYPE               | 296

100 SUPERTYPE             | 156 300
```

```
101 TAN                        | 178
102 THEN                       | 220
103 TO                         | 222
104 TRUE                       | 242
105 TYPE                       | 304
106 TYPEOF                     | 178
107 UNIQUE                     | 165 213 215 237 310
108 UNKNOWN                    | 242
109 UNTIL                      | 312

110 USE                        | 313
111 USEDIN                     | 178
112 VALUE                      | 178
113 VALUE_IN                   | 178
114 VALUE_UNIQUE               | 178
115 VAR                        | 259
116 WHERE                      | 315
117 WHILE                      | 316
118 XOR                        | 158
119 bit                        | 136

120 digit                      | 121 123 127 130 140
121 digits                     | 138 139
122 encoded_character          | 137
123 hex_digit                  | 133
124 letter                     | 127 130 140
125 lparen_not_star            | 142
126 not_lparen_star            | 142
127 not_paren_star             | 126 131 132
128 not_paren_star_quote_special | 129 130 134
129 not_paren_star_special     | 127

130 not_quote                  | 141
131 not_rparen                 | 135
132 not_star                   | 125
133 octet                      | 122
134 special                    |
135 star_not_rparen            | 142
136 binary_literal             | 238
137 encoded_string_literal     | 292
138 integer_literal            | 238
139 real_literal               | 238

140 simple_id                  | 168 187 198 199 210 236 252 260 279 282 305 307 314
141 simple_string_literal      | 292
142 embedded_remark            | 142 143
143 remark                     |
144 tail_remark                | 143
145 attribute_ref              | 169 234 261 266
146 constant_ref               | 186 275
147 entity_ref                 | 195 219 234 245 254 275 278 296 301
148 enumeration_ref            | 200
```

```
149 function_ref             | 207 275

150 parameter_ref            | 216
151 procedure_ref            | 257 275
152 schema_ref               | 267 313
153 type_label_ref           | 306
154 type_ref                 | 200 245 275 309
155 variable_ref             | 216
156 abstract_supertype_declaration | 297
157 actual_parameter_list    | 207 257
158 add_like_op              | 287
159 aggregate_initializer    | 288

160 aggregate_source         | 264
161 aggregate_type           | 211
162 aggregation_types        | 171 309
163 algorithm_head           | 208 258 277
164 alias_stmt               | 291
165 array_type               | 162
166 assignment_stmt          | 291
167 attribute_decl           | 190 203 234
168 attribute_id             | 145 167
169 attribute_qualifier      | 262 263

170 bag_type                 | 162
171 base_type                | 165 170 184 190 203 237 285
172 binary_type              | 289
173 boolean_type             | 289
174 bound_1                  | 176 222
175 bound_2                  | 176 222
176 bound_spec               | 165 170 213 214 215 217 234 237 285
177 built_in_constant        | 186
178 built_in_function        | 207
179 built_in_procedure       | 257

180 case_action              | 182
181 case_label               | 180
182 case_stmt                | 291
183 compound_stmt            | 291
184 constant_body            | 185
185 constant_decl            | 163 280
186 constant_factor          | 261
187 constant_id              | 146 184 270
188 constructed_types        | 309
189 declaration              | 163 280

190 derived_attr             | 191
191 derive_clause            | 194
192 domain_rule              | 315
193 element                  | 159
194 entity_body              | 196
195 entity_constructor       | 288
```

```
196 entity_decl              | 189
197 entity_head              | 196
198 entity_id                | 147 197 246 270
199 enumeration_id           | 148 201

200 enumeration_reference    | 288
201 enumeration_type         | 188
202 escape_stmt              | 291
203 explicit_attr            | 194
204 expression               | 166 181 184 190 193 195 240 241 251 276 283 288
205 factor                   | 303
206 formal_parameter         | 209 259
207 function_call            | 261
208 function_decl            | 189
209 function_head            | 208

210 function_id              | 149 209 270
211 generalized_types        | 253
212 general_aggregation_types | 211
213 general_array_type       | 212
214 general_bag_type         | 212
215 general_list_type        | 212
216 general_ref              | 164 166 261
217 general_set_type         | 212
218 generic_type             | 211
219 group_qualifier          | 262 263

220 if_stmt                  | 291
221 increment                | 222
222 increment_control        | 271
223 index                    | 224 225
224 index_1                  | 226
225 index_2                  | 226
226 index_qualifier          | 263
227 integer_type             | 289
228 interface_specification  | 280
229 interval                 | 288

230 interval_high            | 229
231 interval_item            | 229
232 interval_low             | 229
233 interval_op              | 229
234 inverse_attr             | 235
235 inverse_clause           | 194
236 label                    | 192 311
237 list_type                | 162
238 literal                  | 256
239 local_decl               | 163

240 local_variable           | 239
241 logical_expression       | 192 220 264 312 316
242 logical_literal          | 238
```

```
243 logical_type             | 289
244 multiplication_like_op   | 303
245 named_types              | 171 246 253 284
246 named_type_or_rename     | 313
247 null_stmt                | 291
248 number_type              | 289
249 numeric_expression       | 174 175 221 223 255 273 317

250 one_of                   | 301
251 parameter                | 157
252 parameter_id             | 150 206
253 parameter_type           | 161 206 209 213 214 215 217 240
254 population               | 261
255 precision_spec           | 265
256 primary                  | 288
257 procedure_call_stmt      | 291
258 procedure_decl           | 189
259 procedure_head           | 258

260 procedure_id             | 151 259 270
261 qualifiable_factor       | 256
262 qualified_attribute      | 167 266
263 qualifier                | 164 166 256
264 query_expression         | 288
265 real_type                | 289
266 referenced_attribute     | 311
267 reference_clause         | 228
268 rel_op                   | 269
269 rel_op_extended          | 204

270 rename_id                | 274
271 repeat_control           | 272
272 repeat_stmt              | 291
273 repetition               | 193
274 resource_or_rename       | 267
275 resource_ref             | 274
276 return_stmt              | 291
277 rule_decl                | 280
278 rule_head                | 277
279 rule_id                  | 278

280 schema_body              | 281
281 schema_decl              | 302
282 schema_id                | 152 281
283 selector                 | 182
284 select_type              | 188
285 set_type                 | 162
286 sign                     | 139
287 simple_expression        | 160 204 230 231 232 249
288 simple_factor            | 205
289 simple_types             | 171 253 309
```

```
290 skip_stmt               | 291
291 stmt                    | 164 180 182 183 208 220 258 272 277
292 string_literal          | 238
293 string_type             | 289
294 subsuper                | 197
295 subtype_constraint      | 156 300
296 subtype_declaration     | 294
297 supertype_constraint    | 294
298 supertype_expression    | 250 295 301
299 supertype_factor        | 298

300 supertype_rule          | 297
301 supertype_term          | 299
302 syntax                  |
303 term                    | 287
304 type_decl               | 189
305 type_id                 | 154 246 270 304
306 type_label              | 161 218
307 type_label_id           | 153 306
308 unary_op                | 288
309 underlying_type         | 304

310 unique_clause           | 194
311 unique_rule             | 310
312 until_control           | 271
313 use_clause              | 228
314 variable_id             | 155 164 222 240 264
315 where_clause            | 194 277 304
316 while_control           | 271
317 width                   | 318
318 width_spec              | 172 293
```

Annex B
(normative)
Determination of the allowed entity instantiations

For a particular subtype/supertype graph there may be a large number of complex entity data types and simple entity data types which may be instantiated. This annex will show how these are identified for a general subtype/supertype graph declaration.

NOTE – To illustrate some of the following, consider the set of natural numbers $[1, 2, 3, \ldots]$. This set may be partitioned in many different ways:

- Even numbers and odd numbers: $[2, 4, 6, \ldots]$ and $[1, 3, 5, 7, \ldots]$
- Prime numbers: $[2, 3, 5, 7, \ldots]$
- Numbers divisible by 3: $[3, 6, 9, 12, \ldots]$
- Numbers divisible by 4: $[4, 8, 12, 16, \ldots]$

In *EXPRESS* terms, the natural numbers could be modelled as a supertype and the other sets of numbers as subtypes. It can readily be seen that while some of these subtypes are non-overlapping (e.g. even numbers and odd numbers) others do have some members in common (e.g. the subtypes 'Numbers divisible by 3' and 'Numbers divisible by 4' have the members $[12, 24, \ldots]$ in common).

B.1 Formal approach

Any group of entity data types related by subtype/supertype relationships may be considered as a potentially instantiable subtype/supertype graph. These groups of entity data types are called partial complex entity data types in this formal approach. A partial complex entity data type may contain a single entity data type since a single entity data type is potentially instantiable. A partial complex entity data type is denoted by the names of the entity data types composing it, separated by the & character. Partial complex entity data types may be combined to form other partial complex entity data types. A partial complex entity data type may be a complex entity data type, but full evaluation is required before this is known for certain. A complex entity instance is an instance of a complex entity data type and represents an object of interest.

The following identities hold for any partial complex entity data type:

- $A\&A \equiv A$

  i.e., a particular entity data type will occur only once within a given partial complex entity data type.

- $A\&B \equiv B\&A$

  i.e., the grouping of partial complex entity data types is commutative.

- $A\&(B\&C) \equiv (A\&B)\&C \equiv A\&B\&C$

  i.e., the grouping of partial complex entity data types is associative; the parentheses indicate evaluation precedence which in this case makes no difference to the evaluation.

An evaluated set is defined to be a mathematical set of partial complex entity data types which is denoted by partial complex entity data types separated by ',' enclosed within square brackets. An empty evaluated set is denoted by [].

Two operators can be used to combine a partial complex entity data type and an evaluated set:

– $A + [B1, B2] \equiv [B1, B2] + A \equiv [A, B1, B2]$

The + operator adds the partial complex entity data type to the evaluated set as a new member of the evaluated set. The same partial complex entity data type shall not occur more than once within a single evaluated set.

– $A\&[B1, B2] \equiv [B1, B2]\&A \equiv [A\&B1, A\&B2]$

The & operator adds the patrial complex entity data type to all the partial complex entity data types within the evaluated set. It is therefore distributive over evaluated sets.

Evaluated sets may be combined by the same two mechanisms:

– $[A1, A2] + [B1, B2] \equiv [A1, A2, B1, B2]$

An evaluated set can be formed which contains all of the elements of two combined sets. This is the union of the two sets.

– $[A1, A2]\&[B1, B2] \equiv [A1\&B1, A1\&B2, A2\&B1, A2\&B2]$

An evaluated set can be formed by repeated application of the distribution rule for & for each element of the first evaluated set over the second evaluated set.

Evaluated sets may be filtered using the / operator to create a new evaluated set:

– $[A, A\&B, A\&C, A\&B\&D, B\&C, D]/A \equiv [A, A\&B, A\&C, A\&B\&D]$

The new evaluated set contains only those elements in the original evaluated set which contain the given partial complex entity data type.

– $[A, A\&B, A\&C, A\&B\&D, B\&C, D]/[B, D] \equiv [A\&B, A\&B\&D, B\&C, D]$

The new evaluated set can be formed by repeated filtering of the first evaluated set by each partial complex entity data type in the second evaluated set and combining the results using +.

Evaluated sets may be differenced using the - operator to create a new evaluated set:

– $[A1, A2, B1, B2] - [A2, B1] \equiv [A1, B2]$

An evaluated set can be formed which contains all the elements in the first evaluated set except for those in the second evaluated set.

The following identities hold for any evaluated set:

– $[A, B] \equiv [B, A]$

Evaluated sets are order-independent.

– $[A, A, B] \equiv [A, B]$

A particular complex entity data type will appear only once in any given evaluated set.

– $[A, [B, C]] \equiv [A, B, C]$

Evaluated sets may be nested.

B.2 Supertype operators

Using the above formalism, it is possible to rewrite an *EXPRESS* supertype expression in terms of evaluated sets. The reductions described in B.2.1 to B.2.3 are applied recursively until no supertype term (ONEOF, AND or ANDOR) remains.

These reductions alone do not completely describe the full meaning of the supertype operators, in particular the ONEOF. This requires the full algorithm given in B.3.

B.2.1 ONEOF

The ONEOF list reduces to an evaluated set containing the ONEOF selections, i.e.,

ONEOF$(A, B, \ldots) \longrightarrow [A, B, \ldots]$

B.2.2 AND

The AND operator is equivalent to the & operator and acts upon partial complex entity data types or evaluated sets to produce a partial complex entity data type or an evaluated set.

A AND $B \longrightarrow [A\&B]$
A AND ONEOF$(B1, B2) \longrightarrow A\&[B1, B2] = [A\&B1, A\&B2]$
ONEOF$(A1, A2)$ AND ONEOF$(B1, B2) \longrightarrow [A1, A2]\&[B1, B2] =$
$[A1\&B1, A1\&B2, A2\&B1, A2\&B2]$

B.2.3 ANDOR

The ANDOR operator generates an evaluated set which contains each of the operands separately and the operands combined using the & operator. ANDOR acts upon partial complex entity data types or evaluated sets.

A ANDOR $B \longrightarrow [A, B, A\&B]$
A ANDOR ONEOF$(B1, B2) \longrightarrow [A, [B1, B2], A\&[B1, B2]] = [A, B1, B2, A\&B1, A\&B2]$
ONEOF$(A1, A2)$ ANDOR ONEOF$(B1, B2) \longrightarrow [[A1, A2], [B1, B2], [A1, A2]\&[B1, B2]] =$
$[A1, A2, B1, B2, A1\&B1, A1\&B2, A2\&B1, A2\&B2]$

B.2.4 Precedence of operators

The evaluation of evaluated sets proceeds from left to right, with the highest precedence being evaluated first as specified in 9.2.4.5.

EXAMPLE 154 – The expression below evaluates as follows:
A ANDOR B AND $C \longrightarrow [A, [B\&C], A\&[B\&C]] = [A, B\&C, A\&B\&C]$

B.3 Interpreting the possible complex entity data types

The interpretation of the supertype expression, with additional information available from the declared structure, allows the developer of an *EXPRESS* schema to determine the complex entity

data types which would be instantiable given those declarations. To enable this determination the evaluated set of complex entity data types for the subtype/supertype graph may be generated. For this purpose the following terms are defined:

Multiply inheriting subtype: A multiply inheriting subtype is one which identifies two or more supertypes in its subtype declaration.

Root supertype: A root supertype is a supertype that is not a subtype.

The evaluated set R of complex entity data types is computed by the following process:

a) Identify all entity declarations which form the subtype/supertype graph.

NOTE 1 – This may require multiple iterations in cases with complex subtype/supertype graphs.

b) For each supertype in the subtype/supertype graph, generate the full supertype expression, including implicit ANDOR constraints with subtypes which are otherwise not mentioned in the supertype expression.

c) For each supertype i in the subtype/supertype graph, generate the evaluated set which represents the constraints between its immediate subtypes by applying the reductions in B.2 and the identities in B.1 to the full supertype expression as given by step (b). Combine i with the result using &. If i is not an ABSTRACT supertype, add i to the result using +. Call this set E_i.

d) For each root supertype $r$in the subtype/supertype graph, expand E_r as follows:

(1) For each subtype s of r, replace every occurrence of s (including those within partial complex entity data types) in E_r with E_s, if available, and apply reductions in B.2.

(2) Recursively apply step (d1) to each s, expanding subtypes of s until leaf entities are reached (for which no E_s is available).

NOTE 2 – This recursion must terminate, since there are no cycles in the subtype/supertype graph.

e) Combine root sets. Create $R = \sum_r E_r \equiv E_{r1} + E_{r2} + \ldots$, i.e. R is the union of the sets produced in d.

f) For each multiply inheriting subtype m, do the following:

(1) For each of its immediate supertypes s, generate the set $R/m/s$ which contains exactly those complex data types in R which include both m and s.

(2) Generate the evaluated set of supertype combinations permitted by m, $P_m = R/m/s1\&R/m/s2\& \ldots$, i.e., combine the evaluated sets produced in step (f1) using &.

(3) Generate the evaluated set of supertype combinations which may not include all the supertypes of m, $X_m = \sum_s R/m/s$, i.e., union together the evaluated sets produced in step (f1).

(4) Put $R = (R - X_m) + P_m$.

g) For each supertype term k of the form ONEOF$(S_1, S_2, \ldots)$, do the following.

(1) For each pair of subexpressions S_i, S_j controlled by k $(i < j)$, compute the set of combinations disallowed by ONEOF(S_i, S_j): $D_k^{i,j} = [S_i \& S_j]$. Reduce $D_k^{i,j}$ according to the reductions in B.2 and the identities in B.1.

(2) Set $D_k = \sum_{i,j} D_k^{i,j}$, i.e. D_k is the union of the sets computed in step (g1).

(3) Put $R = R - (R/D_k)$.

h) For each supertype term k of the form S_1 AND S_2, do the following.

(1) Compute the set of required combinations dictated by k, $Q_k = [S_1 \& S_2]$. Reduce Q_k according to the reductions in B.2 and the identities in B.1.

(2) For each entity data type i named in k, compute the set of invalid entity combinations containing i which are disallowed by k, $D_k^i = R/i - R/(Q_k/i)$.

(3) Set $D_k = \sum_i D_k^i$, i.e., D_k is the union of the sets computed in step (h2).

(4) Put $R = R - D_k$.

i) The final evaluated set R is the evaluated set for the input subtype/supertype graph.

EXAMPLE 155 – In this example, only the entity supertype and subtype declarations are given as this is all the information required to interpret the possible complex entity data types.

```
SCHEMA example;
ENTITY p SUPERTYPE OF (ONEOF(m, f) AND ONEOF(c, a));
ENTITY m SUBTYPE OF (p);
ENTITY f SUBTYPE OF (p);
ENTITY c SUBTYPE OF (p);
ENTITY a ABSTRACT SUPERTYPE OF (ONEOF(l, i)) SUBTYPE OF (p);
ENTITY l SUBTYPE OF (a);
ENTITY i SUBTYPE OF (a);
END_SCHEMA;
```

This is represented by *EXPRESS-G* in figure B.1.

The potential complex entity data types can be determined as follows:

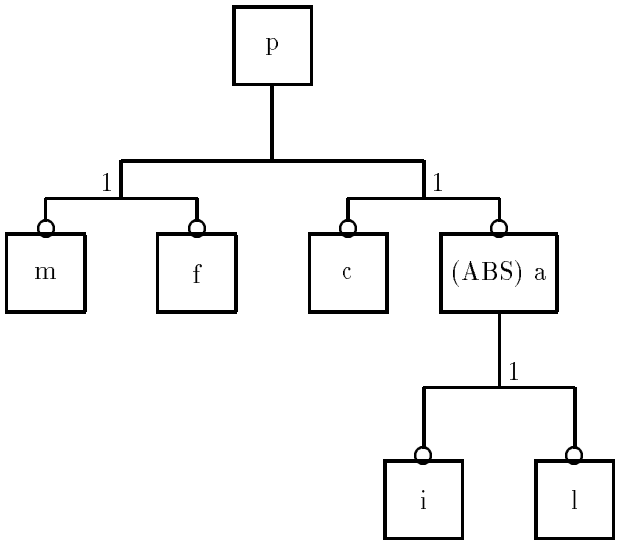


Figure B.1 – EXPRESS-G diagram of schema for example 155.

– The above *EXPRESS* already gives us all of the entity declarations and complete supertype expressions as required in steps (a) and (b).

– Applying step (c) gives:

$$E_p \longrightarrow [p\&m\&c, p\&m\&a, p\&f\&c, p\&f\&a, p]$$
$$E_a \longrightarrow [a\&l, a\&i]$$

– Applying step (d) expands the declarations of the root entities, p. The resulting set is:

$$E_p = [p\&m\&c, p\&m\&a\&l, p\&m\&a\&i, p\&f\&c, p\&f\&a\&l, p\&f\&a\&i, p]$$

– Combining the root sets in step (e) gives:

$$R = [p\&m\&c, p\&m\&a\&l, p\&m\&a\&i, p\&f\&c, p\&f\&a\&l, p\&f\&a\&i, p]$$

– There are no multiply inheriting subtypes, so step (f) is not required.

– Applying step (g) to each ONEOF constraint gives:

- ONEOF(m, f):

$$D_1^{1,2} = [m\&f]$$
$$D_1 = [m\&f]$$

Removing D_1 from R according to step (g3) leaves R unchanged. Thus we are left with:

$$R = [p\&m\&c, p\&m\&a\&l, p\&m\&a\&i, p\&f\&c, p\&f\&a\&l, p\&f\&a\&i, p]$$

- ONEOF(c, a):

$$\begin{aligned} D_2^{1,2} &= [c\&a] \\ D_2 &= [c\&a] \end{aligned}$$

Removing D_2 from R according to step (g3) leaves R unchanged. Thus we are left with:

$$R = [p\&m\&c, p\&m\&a\&l, p\&m\&a\&i, p\&f\&c, p\&f\&a\&l, p\&f\&a\&i, p]$$

- ONEOF(l, i):

$$\begin{aligned} D_3^{1,2} &= [l\&i] \\ D_3 &= [l\&i] \end{aligned}$$

Removing D_3 from R according to step (g3) leaves R unchanged. Thus we are left with:

$$R = [p\&m\&c, p\&m\&a\&l, p\&m\&a\&i, p\&f\&c, p\&f\&a\&l, p\&f\&a\&i, p]$$

– Applying step (h) to each AND constraint gives:

- ONEOF(m, f) AND ONEOF(c, a):

$$\begin{aligned} Q_1 &= [m\&c, m\&a, f\&c, f\&a] \\ D_1^m &= [] \\ D_1^f &= [] \\ D_1^c &= [] \\ D_1^a &= [] \\ D_1 &= [] \end{aligned}$$

Removing D_1 from R according to step (h4) leaves R unchanged. Thus we are left with:

$$R = [p\&m\&c, p\&m\&a\&l, p\&m\&a\&i, p\&f\&c, p\&f\&a\&l, p\&f\&a\&i, p]$$

– According to step (i), the result is thus:

$$R = [p\&m\&c, p\&m\&a\&l, p\&m\&a\&i, p\&f\&c, p\&f\&a\&l, p\&f\&a\&i, p]$$

This example, although apparently arbitrary, could have been made more realistic if the entities had been given more descriptive names. For instance, if instead of `p`, `m`, `f`, `c`, `a`, `l` and `i` the entities were respectively called, `person`, `male`, `female`, `citizen`, `alien`, `legal_alien` and `illegal_alien`.

With this interpretation, reading off a few of the items in the final evaluated set gives:

– A `person`.

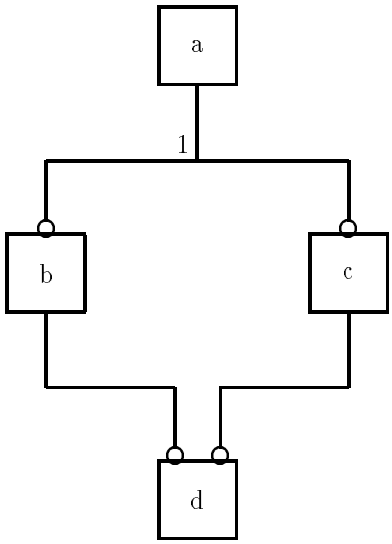


Figure B.2 – EXPRESS-G diagram of schema for example 156.

– A `person` who is `male` and a `citizen`.

– A `person` who is a `male alien` and who is a `illegal_alien`.

– A `person` who ...

EXAMPLE 156 – This example demonstrates that ONEOF is a global constraint which cannot be overridden by multiple inheritance.

```
SCHEMA diamond;
ENTITY a SUPERTYPE OF (ONEOF(b, c));
ENTITY b SUPERTYPE OF (d) SUBTYPE OF (a);
ENTITY c SUPERTYPE OF (d) SUBTYPE OF (a);
ENTITY d SUBTYPE OF (b, c);
END_SCHEMA;
```

This is represented by *EXPRESS-G* in figure B.2.

The potential complex entity data types can be determined as follows:

– The above *EXPRESS* already gives us all of the entity declarations and complete supertype expressions as required in steps (a) and (b).

– Applying step (c) gives:

$$\begin{aligned} E_a &\longrightarrow [a\&b, a\&c, a] \\ E_b &\longrightarrow [b\&d, b] \\ E_c &\longrightarrow [c\&d, c] \\ E_d &\longrightarrow [d] \end{aligned}$$

– Applying step (d) expands the declarations of the root entities, a. The resulting sets are:

$$E_a = [a\&b\&d, a\&b, a\&c\&d, a\&c, a]$$

– Combining the root sets in step (e) gives:

$$R = [a\&b\&d, a\&b, a\&c\&d, a\&c, a]$$

– Applying step (f) to each multiply inheriting subtype gives the following results:

- For entity d:

$$\begin{aligned} C_d^b &= [a\&b\&d] \\ C_d^c &= [a\&c\&d] \end{aligned}$$

$$\begin{aligned} P_d &= [a\&b\&d\&c] \\ X_d &= [a\&b\&d, a\&c\&d] \end{aligned}$$

The new set $R = (R - X_d) + P_d$ is then: $[a\&b, a\&c, a, a\&b\&d\&c]$

– Applying step (g) to each ONEOF constraint gives:

- ONEOF(b, c):

$$\begin{aligned} D_1^{1,2} &= [b\&c] \\ D_1 &= [b\&c] \end{aligned}$$

Removing D_1 from R according to step (g3) removes the following elements from R: $[a\&b\&d\&c]$.

Thus we are left with:

$$R = [a\&b, a\&c, a]$$

– There are no supertype expressions which use AND, so step (h) is not required.

– According to step (i), the result is thus:

$$R = [a\&b, a\&c, a]$$

EXAMPLE 157 – This example shows the effect of applying constraints to a complex structure which contains at least one of each possible type of constraint. This example is not expected to model any useful concept and is used purely to demonstrate the algorithm.

```
SCHEMA complex;
ENTITY a SUPERTYPE OF (ONEOF(b, c) AND d ANDOR f);
```

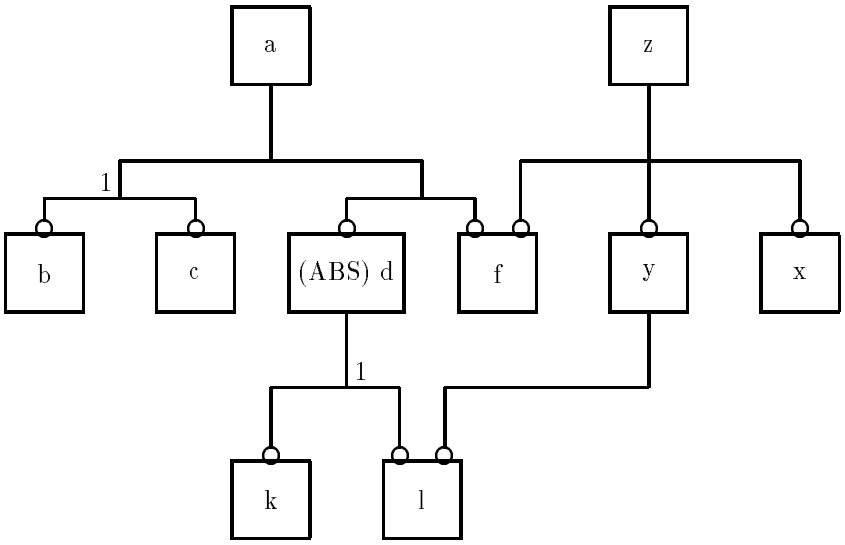


Figure B.3 – EXPRESS-G diagram of schema for example 157.

```
ENTITY b SUBTYPE OF (a);
ENTITY c SUBTYPE OF (a);
ENTITY d ABSTRACT SUPERTYPE OF (ONEOF(k, l)) SUBTYPE OF (a);
ENTITY f SUBTYPE OF (a, z);
ENTITY k SUBTYPE OF (d);
ENTITY l SUBTYPE OF (d, y);
ENTITY x SUBTYPE OF (z);
ENTITY y SUPERTYPE OF (l) SUBTYPE OF (z);
ENTITY z SUPERTYPE OF (f ANDOR x ANDOR y);
END_SCHEMA;
```

This is represented by *EXPRESS-G* in figure B.3.

The potential complex entity data types can be determined as follows:

– The above *EXPRESS* explicitly specifies all of the entity declarations and complete supertype expressions as required in steps (a) and (b).

– Applying step (c) gives:

$$
\begin{aligned}
E_a &\longrightarrow [a, a\&b\&d, a\&b\&d\&f, a\&c\&d, a\&c\&d\&f, a\&f] \\
E_d &\longrightarrow [d\&k, d\&l] \\
E_y &\longrightarrow [l\&y, y] \\
E_z &\longrightarrow [f\&x\&y\&z, f\&x\&z, f\&y\&z, f\&z, x\&y\&z, x\&z, y\&z, z]
\end{aligned}
$$

– Applying step (d) expands the declarations of the root entities, `a`, `z`. The resulting sets are:

$$E_a = [a, a\&b\&d\&k, a\&b\&d\&l, a\&b\&d\&f\&k, a\&b\&d\&f\&l, a\&c\&d\&k, a\&c\&d\&l,$$

$a\&c\&d\&f\&k, a\&c\&d\&f\&l, a\&f]$

$E_z = [f\&l\&x\&y\&z, f\&l\&y\&z, f\&x\&y\&z, f\&x\&z, f\&y\&z, f\&z, l\&x\&y\&z, l\&y\&z, x\&y\&z, x\&z, y\&z, z]$

– Combining the root sets in step (e) gives:

$R = [a, a\&b\&d\&k, a\&b\&d\&l, a\&b\&d\&f\&k, a\&b\&d\&f\&l, a\&c\&d\&k, a\&c\&d\&l, a\&c\&d\&f\&k, a\&c\&d\&f\&l, a\&f, f\&l\&x\&y\&z, f\&l\&y\&z, f\&x\&y\&z, f\&x\&z, f\&y\&z, f\&z, l\&x\&y\&z, l\&y\&z, x\&y\&z, x\&z, y\&z, z]$

– Applying step (f) to each multiply inheriting subtype gives the following results:

- For entity f:

$C_f^a = [a\&b\&d\&k\&f, a\&b\&d\&l\&f, a\&c\&d\&k\&f, a\&c\&d\&l\&f, a\&f]$

$C_f^z = [f\&l\&x\&y\&z, f\&l\&y\&z, f\&x\&y\&z, f\&x\&z, f\&y\&z, f\&z]$

$P_f = [a\&b\&d\&f\&k\&z, a\&b\&d\&f\&k\&x\&z, a\&b\&d\&f\&k\&l\&y\&z, a\&b\&d\&f\&k\&y\&z, a\&b\&d\&f\&k\&l\&x\&y\&z, a\&b\&d\&f\&k\&x\&y\&z, a\&b\&d\&f\&l\&z, a\&b\&d\&f\&l\&x\&z, a\&b\&d\&f\&l\&y\&z, a\&b\&d\&f\&l\&x\&y\&z, a\&c\&d\&f\&k\&z, a\&c\&d\&f\&k\&x\&z, a\&c\&d\&f\&k\&l\&y\&z, a\&c\&d\&f\&k\&y\&z, a\&c\&d\&f\&k\&l\&x\&y\&z, a\&c\&d\&f\&k\&x\&y\&z, a\&c\&d\&f\&l\&z, a\&c\&d\&f\&l\&x\&z, a\&c\&d\&f\&l\&y\&z, a\&c\&d\&f\&l\&x\&y\&z, a\&f\&z, a\&f\&x\&z, a\&f\&l\&y\&z, a\&f\&y\&z, a\&f\&l\&x\&y\&z, a\&f\&x\&y\&z]$

$X_f = [a\&b\&d\&f\&k, a\&b\&d\&f\&l, a\&c\&d\&f\&k, a\&c\&d\&f\&l, a\&f, f\&l\&x\&y\&z, f\&l\&y\&z, f\&x\&y\&z, f\&x\&z, f\&y\&z, f\&z]$

The new set $R = (R - X_f) + P_f$ is then: $[a, a\&b\&d\&f\&k\&z, a\&b\&d\&f\&k\&x\&z, a\&b\&d\&f\&k\&l\&y\&z, a\&b\&d\&f\&k\&y\&z, a\&b\&d\&f\&k\&l\&x\&y\&z, a\&b\&d\&f\&k\&x\&y\&z, a\&b\&d\&f\&l\&z, a\&b\&d\&f\&l\&x\&z, a\&b\&d\&f\&l\&y\&z, a\&b\&d\&f\&l\&x\&y\&z, a\&b\&d\&k, a\&b\&d\&l, a\&c\&d\&f\&k\&z, a\&c\&d\&f\&k\&x\&z, a\&c\&d\&f\&k\&l\&y\&z, a\&c\&d\&f\&k\&y\&z, a\&c\&d\&f\&k\&l\&x\&y\&z, a\&c\&d\&f\&k\&x\&y\&z, a\&c\&d\&f\&l\&z, a\&c\&d\&f\&l\&x\&z, a\&c\&d\&f\&l\&y\&z, a\&c\&d\&f\&l\&x\&y\&z, a\&c\&d\&k, a\&c\&d\&l, a\&f\&z, a\&f\&x\&z, a\&f\&l\&y\&z, a\&f\&y\&z, a\&f\&l\&x\&y\&z, a\&f\&x\&y\&z, l\&x\&y\&z, l\&y\&z, x\&y\&z, x\&z, y\&z, z]$

- For entity l:

$C_l^d = [a\&b\&d\&f\&k\&l\&y\&z, a\&b\&d\&f\&k\&l\&x\&y\&z, a\&b\&d\&f\&l\&z, a\&b\&d\&f\&l\&x\&z, a\&b\&d\&f\&l\&y\&z, a\&b\&d\&f\&l\&x\&y\&z, a\&b\&d\&l, a\&c\&d\&f\&k\&l\&y\&z, a\&c\&d\&f\&k\&l\&x\&y\&z, a\&c\&d\&f\&l\&z, a\&c\&d\&f\&l\&x\&z, a\&c\&d\&f\&l\&y\&z, a\&c\&d\&f\&l\&x\&y\&z, a\&c\&d\&l]$

$C_l^y = [a\&b\&d\&f\&k\&l\&y\&z, a\&b\&d\&f\&k\&l\&x\&y\&z, a\&b\&d\&f\&l\&y\&z, a\&b\&d\&f\&l\&x\&y\&z, a\&c\&d\&f\&k\&l\&y\&z, a\&c\&d\&f\&k\&l\&x\&y\&z, a\&c\&d\&f\&l\&y\&z, a\&c\&d\&f\&l\&x\&y\&z, a\&f\&l\&y\&z, a\&f\&l\&x\&y\&z, l\&x\&y\&z$

$l\&y\&z]$

$$
\begin{aligned}
P_l &= [a\&b\&c\&d\&f\&k\&l\&y\&z, a\&b\&c\&d\&f\&k\&l\&x\&y\&z, a\&b\&c\&f\&l\&y\&z,\\
&\quad a\&b\&c\&f\&l\&x\&y\&z, a\&b\&d\&f\&k\&l\&y\&z, a\&b\&d\&f\&k\&l\&x\&y\&z,\\
&\quad a\&b\&d\&f\&l\&y\&z, a\&b\&d\&f\&l\&x\&y\&z, a\&b\&d\&l\&x\&y\&z, a\&b\&d\&l\&y\&z,\\
&\quad a\&c\&d\&f\&k\&l\&y\&z, a\&c\&d\&f\&k\&l\&x\&y\&z, a\&c\&d\&f\&l\&y\&z,\\
&\quad a\&c\&d\&f\&l\&x\&y\&z, a\&c\&d\&l\&x\&y\&z, a\&c\&d\&l\&y\&z]\\
X_l &= [a\&b\&d\&f\&k\&l\&y\&z, a\&b\&d\&f\&k\&l\&x\&y\&z, a\&b\&d\&f\&l\&z,\\
&\quad a\&b\&d\&f\&l\&x\&z, a\&b\&d\&f\&l\&y\&z, a\&b\&d\&f\&l\&x\&y\&z, a\&b\&d\&l,\\
&\quad a\&c\&d\&f\&k\&l\&y\&z, a\&c\&d\&f\&k\&l\&x\&y\&z, a\&c\&d\&f\&l\&z, a\&c\&d\&f\&l\&x\&z,\\
&\quad a\&c\&d\&f\&l\&y\&z, a\&c\&d\&f\&l\&x\&y\&z, a\&c\&d\&l, a\&f\&l\&y\&z, a\&f\&l\&x\&y\&z,\\
&\quad l\&x\&y\&z, l\&y\&z]
\end{aligned}
$$

The new set $R = (R - X_l) + P_l$ is then: $[a, a\&b\&c\&d\&f\&k\&l\&y\&z,$
$a\&b\&c\&d\&f\&k\&l\&x\&y\&z, a\&b\&c\&f\&l\&y\&z, a\&b\&c\&f\&l\&x\&y\&z,$
$a\&b\&d\&f\&k\&l\&y\&z, a\&b\&d\&f\&k\&l\&x\&y\&z, a\&b\&d\&f\&k\&x\&z, a\&b\&d\&f\&k\&y\&z,$
$a\&b\&d\&f\&k\&x\&y\&z, a\&b\&d\&f\&k\&z, a\&b\&d\&f\&l\&y\&z, a\&b\&d\&f\&l\&x\&y\&z,$
$a\&b\&d\&k, a\&b\&d\&l\&x\&y\&z, a\&b\&d\&l\&y\&z, a\&c\&d\&f\&k\&l\&x\&y\&z,$
$a\&c\&d\&f\&k\&l\&y\&z, a\&c\&d\&f\&l\&y\&z, a\&c\&d\&f\&l\&x\&y\&z, a\&c\&d\&f\&k\&x\&z,$
$a\&c\&d\&f\&k\&y\&z, a\&c\&d\&f\&k\&x\&y\&z, a\&c\&d\&f\&k\&z, a\&c\&d\&k,$
$a\&c\&d\&l\&x\&y\&z, a\&c\&d\&l\&y\&z, a\&f\&z, a\&f\&x\&z, a\&f\&y\&z, a\&f\&x\&y\&z,$
$x\&y\&z, x\&z, y\&z, z]$

– Applying step (g) to each ONEOF constraint gives:

  • ONEOF(b, c):

$$
\begin{aligned}
D_1^{1,2} &= [b\&c]\\
D_1 &= [b\&c]
\end{aligned}
$$

Removing D_1 from R according to step (g3) removes the following elements from R:
$[a\&b\&c\&d\&f\&k\&l\&y\&z, a\&b\&c\&d\&f\&k\&l\&x\&y\&z, a\&b\&c\&f\&l\&y\&z,$
$a\&b\&c\&f\&l\&x\&y\&z]$

Thus we are left with:

$$
\begin{aligned}
R &= [a, a\&b\&d\&f\&k\&l\&y\&z, a\&b\&d\&f\&k\&l\&x\&y\&z, a\&b\&d\&f\&k\&x\&z,\\
&\quad a\&b\&d\&f\&k\&y\&z, a\&b\&d\&f\&k\&x\&y\&z, a\&b\&d\&f\&k\&z, a\&b\&d\&f\&l\&y\&z,\\
&\quad a\&b\&d\&f\&l\&x\&y\&z, a\&b\&d\&k, a\&b\&d\&l\&x\&y\&z, a\&b\&d\&l\&y\&z,\\
&\quad a\&c\&d\&f\&k\&l\&x\&y\&z, a\&c\&d\&f\&k\&l\&y\&z, a\&c\&d\&f\&l\&y\&z,\\
&\quad a\&c\&d\&f\&l\&x\&y\&z, a\&c\&d\&f\&k\&x\&z, a\&c\&d\&f\&k\&y\&z,\\
&\quad a\&c\&d\&f\&k\&x\&y\&z, a\&c\&d\&f\&k\&z, a\&c\&d\&k, a\&c\&d\&l\&x\&y\&z,\\
&\quad a\&c\&d\&l\&y\&z, a\&f\&z, a\&f\&x\&z, a\&f\&y\&z, a\&f\&x\&y\&z, x\&y\&z, x\&z, y\&z, z]
\end{aligned}
$$

  • ONEOF(k, l):

$$
\begin{aligned}
D_2^{1,2} &= [k\&l]\\
D_2 &= [k\&l]
\end{aligned}
$$

Removing D_2 from R according to step (g3) removes the following elements from R:
$[a\&b\&d\&f\&k\&l\&y\&z, a\&b\&d\&f\&k\&l\&x\&y\&z, a\&c\&d\&f\&k\&l\&y\&z,$
$a\&c\&d\&f\&k\&l\&x\&y\&z]$

Thus we are left with:

$$\begin{aligned} R = [&a, a\&b\&d\&f\&k\&x\&z, a\&b\&d\&f\&k\&y\&z, a\&b\&d\&f\&k\&x\&y\&z,\\ &a\&b\&d\&f\&k\&z, a\&b\&d\&f\&l\&y\&z, a\&b\&d\&f\&l\&x\&y\&z, a\&b\&d\&k,\\ &a\&b\&d\&l\&x\&y\&z, a\&b\&d\&l\&y\&z, a\&c\&d\&f\&l\&y\&z, a\&c\&d\&f\&l\&x\&y\&z,\\ &a\&c\&d\&f\&k\&x\&z, a\&c\&d\&f\&k\&y\&z, a\&c\&d\&f\&k\&x\&y\&z, a\&c\&d\&f\&k\&z,\\ &a\&c\&d\&k, a\&c\&d\&l\&x\&y\&z, a\&c\&d\&l\&y\&z, a\&f\&z, a\&f\&x\&z, a\&f\&y\&z,\\ &a\&f\&x\&y\&z, x\&y\&z, x\&z, y\&z, z] \end{aligned}$$

– Applying step (h) to each AND constraint gives:

• ONEOF(b, c) ANDd:

$$\begin{aligned} Q_1 &= [b\&d, c\&d]\\ D_1^b &= []\\ D_1^c &= []\\ D_1^d &= []\\ D_1 &= [] \end{aligned}$$

Removing D_1 from R according to step (h4) leaves R unchanged. Thus we are left with:

$$\begin{aligned} R = [&a, a\&b\&d\&f\&k\&l\&y\&z, a\&b\&d\&f\&k\&l\&x\&y\&z, a\&b\&d\&f\&k\&x\&z,\\ &a\&b\&d\&f\&k\&y\&z, a\&b\&d\&f\&k\&x\&y\&z, a\&b\&d\&f\&k\&z, a\&b\&d\&f\&l\&y\&z,\\ &a\&b\&d\&f\&l\&x\&y\&z, a\&c\&d\&f\&k\&l\&y\&z, a\&b\&d\&k, a\&b\&d\&l\&x\&y\&z,\\ &a\&b\&d\&l\&y\&z, a\&c\&d\&f\&k\&l\&x\&y\&z, a\&c\&d\&f\&l\&y\&z, a\&c\&d\&f\&l\&x\&y\&z,\\ &a\&c\&d\&f\&k\&x\&z, a\&c\&d\&f\&k\&y\&z, a\&c\&d\&f\&k\&x\&y\&z, a\&c\&d\&f\&k\&z,\\ &a\&c\&d\&k, a\&c\&d\&l\&x\&y\&z, a\&c\&d\&l\&y\&z, a\&f\&z, a\&f\&x\&z, a\&f\&y\&z,\\ &a\&f\&x\&y\&z, x\&y\&z, x\&z, y\&z, z] \end{aligned}$$

– According to step (i), the result is thus:

$$\begin{aligned} R = [&a, a\&b\&d\&f\&k\&l\&y\&z, a\&b\&d\&f\&k\&l\&x\&y\&z, a\&b\&d\&f\&k\&x\&z,\\ &a\&b\&d\&f\&k\&y\&z, a\&b\&d\&f\&k\&x\&y\&z, a\&b\&d\&f\&k\&z, a\&b\&d\&f\&l\&y\&z,\\ &a\&b\&d\&f\&l\&x\&y\&z, a\&c\&d\&f\&k\&l\&y\&z, a\&b\&d\&k, a\&b\&d\&l\&x\&y\&z,\\ &a\&b\&d\&l\&y\&z, a\&c\&d\&f\&k\&l\&x\&y\&z, a\&c\&d\&f\&l\&y\&z, a\&c\&d\&f\&l\&x\&y\&z,\\ &a\&c\&d\&f\&k\&x\&z, a\&c\&d\&f\&k\&y\&z, a\&c\&d\&f\&k\&x\&y\&z, a\&c\&d\&f\&k\&z,\\ &a\&c\&d\&k, a\&c\&d\&l\&x\&y\&z, a\&c\&d\&l\&y\&z, a\&f\&z, a\&f\&x\&z, a\&f\&y\&z,\\ &a\&f\&x\&y\&z, x\&y\&z, x\&z, y\&z, z] \end{aligned}$$

Annex C
(normative)
Instance limits imposed by the interface specification

When complex subtype/supertype graphs are interfaced, the valid complex entity data types are computed by extending the rules given in clause 11 and annex B. By specifying only those entities which are required in the current schema, a subtype/supertype graph defined in one or more other schemas may pruned for use within the current schema.

This annex gives the rules required to resolve the subtype/supertype graphs where one or more entity data types, originally in the graph, have not been interfaced. These missing entity data types leave holes in the supertype expressions. Such holes are denoted by <> in this clause. The following reductions are used to remove these holes from a supertype expression:

- ONEOF(A, <>, ...) $\longrightarrow$ ONEOF(A, ...)
- ONEOF(<>) $\longrightarrow$ <>
- ONEOF(A) $\longrightarrow$ A
- A AND <> $\longrightarrow$ ONEOF(A, A)
- A ANDOR <> $\longrightarrow$ A

The treatment of AND is to ensure that entity data types which are constrained in the original schema to be combined, are not allowed to exist in this schema (ensured by ONEOF(A,A)) if the entity data types they were to be combined with are not interfaced.

The evaluated set of valid complex entity data types for a schema which interfaces with other schemas is computed by the following algorithm:

a) Generate the complete entity pool for the current schema. The complete entity pool consists of the following:

(1) all entities defined in the current schema;

(2) all entities USE'd or REFERENCE'd into the current schema;

(3) all entities implicitly interfaced into the current schema.

NOTE 3 – The complete entity pool may contain more than one entity with the same name (in the case of implicit references from multiple schemas), or it may include the same entity under different names (in the case of USE FROM ... AS). In the former case, the entity pool contains each of the identically-named entities, while in the latter case the entity pool contains only the single entity, despite its multiple names.

b) For each supertype in the entity pool, prune the supertype expression to remove all references to entities not in the entity pool. Repeatedly apply the reductions above to remove

the holes thus introduced, producing a valid supertype expression referring only to entities in the entity pool.

c) For each supertype in the entity pool, write out the full supertype expression, including implicit ANDOR constraints with subtypes which are otherwise not mentioned in the supertype expression.

NOTE 4 – The implicit ANDOR constraints include any subtypes added in schemas other than the declaring schema, as well as those in the declaring schema which do not otherwise appear in the supertype expression.

d) Compute the evaluated set according to the algorithm in annex B, starting with step (c).

A complex entity data type in the evaluated set which contains at least one locally declared or USE'd entity may be independently instantiated. A complex entity data type which does not contain any such entity cannot be independently instantiated in the current schema.

NOTE 5 – If there is an explicitly interfaced entity which is not contained in any complex entity data type in the resultant evaluated set, this entity cannot be instantiated at all. It is likely that such an entity has been interfaced in error.

EXAMPLES

158 – The schema **example** (see example 155 in annex B) is used to demonstrate the algorithm.

```
SCHEMA test;
USE FROM example (l);
REFERENCE FROM example (m);
END_SCHEMA;
```

The potential complex entity data types are determined as follows:

– The entity pool is l, m, a, p: l and m are explicitly interfaced, a and p are implicitly interfaced because they are in the supertype chain of l.

– Pruning the supertype expression for p and reducing it (step b) gives:

$$\begin{aligned} &\text{ONEOF}(m, f) \text{ AND ONEOF}(c, a) \\ &\text{ONEOF}(m, <>) \text{ AND ONEOF}(<>, a) \\ &\text{ONEOF}(m) \text{ AND ONEOF}(a) \\ &m \text{ AND } a \end{aligned}$$

Similarly, for a we find:

$$\begin{aligned} &\text{ONEOF}(l, i) \\ &\text{ONEOF}(l, <>) \\ &\text{ONEOF}(l) \\ &l \end{aligned}$$

– In this case, the supertype expressions are already complete, as required by step (c).

– Applying the evaluated set algorithm in step (d) gives the evaluated set $R = [p, a\&l\&m\&p]$. The complex entity data type $a\&l\&m\&p$ contains the explicitly USE'd entity l, and so can be independently instantiated. p, on the other hand, cannot be independently instantiated in the current schema.

159 – Assuming the following schemas are available:

```
SCHEMA s1;
ENTITY e1 SUPERTYPE OF (e11 ANDOR e12); END_ENTITY;
ENTITY e11 SUBTYPE OF (e1); END_ENTITY;
ENTITY e12 SUBTYPE OF (e1); END_ENTITY;
END_SCHEMA;

SCHEMA s2;
USE FROM s1 (e11 AS f);
ENTITY e211 SUBTYPE OF (f); END_ENTITY;
ENTITY e212 SUBTYPE OF (f); END_ENTITY;
END_SCHEMA;

SCHEMA s3;
USE FROM s1 (e12 as g);
ENTITY e321 SUBTYPE OF (g); END_ENTITY;
ENTITY e322 SUBTYPE OF (g); END_ENTITY;
END_SCHEMA;
```

The evaluated sets for these schemas are as follows:

s1 $[e1, e1\&e11, e1\&e12, e1\&e11\&e12]$

s2 $[e1\&f, e1\&f\&e211, e1\&f\&e212, e1\&f\&e211\&e212]$

s3 $[e1\&g, e1\&g\&e321, e1\&g\&e322, e1\&g\&e321\&e322]$

If schema `test` is defined as below:

```
SCHEMA test;
USE FROM s2 (e211);
USE FROM s3 (e322);
END_SCHEMA;
```

The potential complex entity data types are determined as follows:

– The entity pool is $e211, e322, f, g, e1$: $e211$ and $e322$ are explicitly interfaced, f, g and $e1$ are implicitly interfaced because they are in the supertype chain of $e211$ and $e322$. f and g are renames of $e11$ and $e12$ respectively, so $e11$ and $e12$ are effectively members of the entity pool.

– Pruning the supertype expression for $e1$ and reducing it (step b) gives:

$$e11 \text{ ANDOR } e12$$

$$f \text{ ANDOR } g$$

for f we find:

$$e211 \text{ ANDOR } e212$$
$$e211 \text{ ANDOR } <>$$
$$e211$$

for g we find:

$$e321 \text{ ANDOR } e322$$
$$<> \text{ ANDOR } e322$$
$$e322$$

– In this case, the supertype expressions are already complete, as required by step (c).

– Applying the evaluated set algorithm in step (d) gives the evaluated set $R = [e1, e1\&f, e1\&g, e1\&f\&g, e1\&f\&e211, e1\&f\&g\&e211, e1\&g\&e322, e1\&f\&g\&e322, e1\&f\&g\&e211\&e322]$. The complex entity data types $e1\&f\&e211, e1\&f\&g\&e211, e1\&g\&e322, e1\&f\&g\&e322, e1\&f\&g\&e211\&e322$ contains one of the explicitly USE'd entity $e211$ or $e322$, and so can be independently instantiated. $e1, e1\&f, e1\&g, e1\&f\&g$, on the other hand, cannot be independently instantiated in the current schema.

Annex D
(normative)
EXPRESS-G: A graphical subset of EXPRESS

D.1 Introduction and overview

EXPRESS-G is a formal graphical notation for the display of data specifications defined in the *EXPRESS* language. The notation supports a subset of the *EXPRESS* language.

EXPRESS-G supports the following:

– various levels of data abstraction;

– diagrams spanning more than one page;

– diagrams using minimal computer graphics capabilities, including using only non-graphic symbols.

EXPRESS-G is represented by graphic symbols forming a diagram. The notation has three types of symbols:

Definition: symbols denoting simple data types, named data types, constructed data types and schema declarations;

Relationship: symbols describing relationships which exist among the definitions;

Composition: symbols enabling a diagram to be displayed on more than one page.

EXPRESS-G supports simple data types, named data types, relationships and cardinality. *EXPRESS-G* also supports the notation for one or more schemas. It does not provide any support for the constraint mechanisms provided by the *EXPRESS* language.

NOTE – *EXPRESS-G* may be used as a data specification language in its own right i.e., there is no requirement to have an associated *EXPRESS* specification.

EXAMPLE 160 – The figure D.1 and figure D.2 show an *EXPRESS-G* diagram for the single *EXPRESS* schema given in example 171 (see annex H). The graphical diagram is divided to illustrate the use of multiple pages.

A `person` has certain defining characteristics, including a first name, a last name, an optional nickname, date of birth, and a description of their hair. A `person` is either `male` or `female`. A `male` may have a `female` wife; in which case the `female` has a `male` husband. A person may have children who are also `persons`.

D.2 Definition symbols

The definitions of data types and schemas within a diagram are denoted by boxes which enclose the name of the item being defined. The relationships between the items are denoted by the

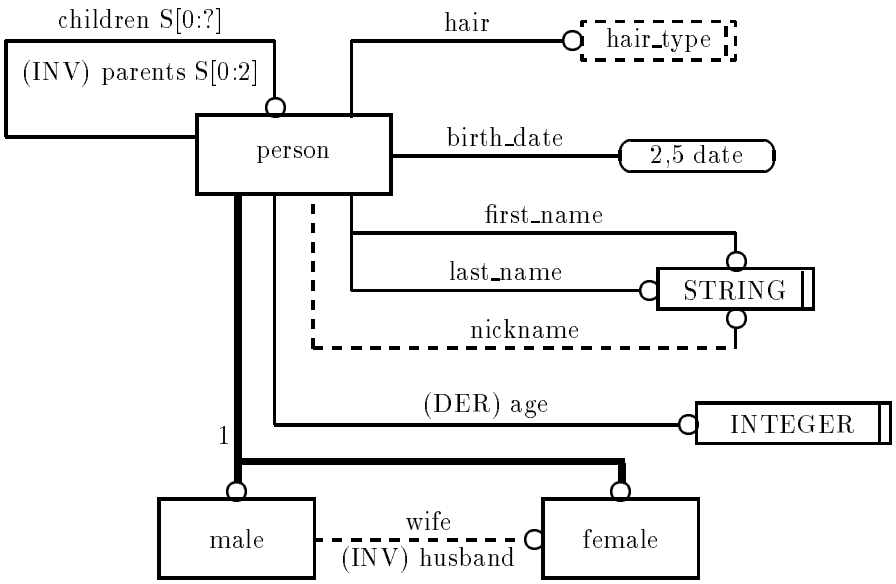


Figure D.1 – Complete entity level diagram of example 171 (Page 1 of 2).

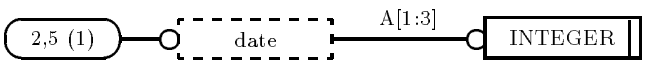


Figure D.2 – Complete entity level diagram of example 171 (Page 2 of 2).

lines joining the boxes. Differing line styles provide information on the kind of definition or relationship.

D.2.1 Symbol for simple data types

The symbol for an *EXPRESS* simple data type is a rectangular solid box with a double vertical line at the right end of the box. The name of the data type is enclosed within the box, as shown in figure D.3.

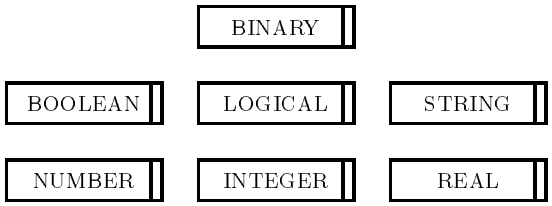


Figure D.3 – Symbols for EXPRESS simple data types.

D.2.2 Symbols for constructed data types

The symbols for the constructed data types of *EXPRESS*, SELECT and ENUMERATION, are dashed boxes. The name of the data type is enclosed within the box as shown in figure D.4.

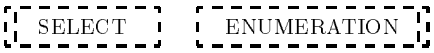


Figure D.4 – Symbols for EXPRESS constructed data types.

The symbol for a SELECT data type consists of a dashed box with a double vertical line at the left.

The symbol for an ENUMERATION data type consists of a dashed box with a double vertical line at the right. *EXPRESS-G* does not provide for the representation of the enumerated list.

NOTE – The ENUMERATION data type symbol resembles the simple data type symbols, having a second vertical bar at the right, since simple and ENUMERATION data types are atomic data types in *EXPRESS-G*.

EXPRESS only allows the SELECT and ENUMERATION data types to be used as a representation of a defined data type. *EXPRESS-G* provides an abbreviated notation whereby the defined data type name is written within the dashed box representing the SELECT or ENUMERATION, instead of the data type name, and the defined data type symbol is not given, as seen in figure D.5 (see D.5.4).

EXAMPLE 161 – The two diagrams in figure D.6 are equivalent:

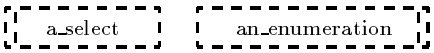


Figure D.5 – Abbreviated symbols for the EXPRESS constructed data types when used as the representation of defined data types.

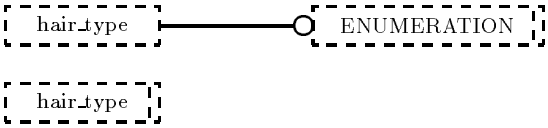


Figure D.6 – Example of alternative methods for representing an ENUMERATION

An implementation of an *EXPRESS-G* editing tool may use the full form of constructed data type representation, the abbreviated form or both. The implementor of an *EXPRESS-G* editing tool shall indicate which of these forms is used using annex E.

D.2.3 Symbols for defined data types

The symbol for a defined data type consists of a dashed box enclosing the name of the TYPE, as shown in figure D.7.

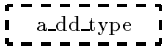


Figure D.7 – Symbols for EXPRESS defined data type.

D.2.4 Symbols for entity data types

The symbol for an entity data type consists of a solid box enclosing the name of the ENTITY, as shown in figure D.8.

D.2.5 Symbols for functions and procedures

EXPRESS-G does not support any notation for either FUNCTION or PROCEDURE definitions.

D.2.6 Symbols for rules

EXPRESS-G does not support any notation for a RULE definition. The names of entities that are parameters in a RULE may be flagged with an asterisk (see D.5.3).

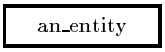


Figure D.8 – Symbol for an EXPRESS entity data type.

D.2.7 Symbols for schemas

The symbol for a SCHEMA (figure D.9) is a rectangular solid box with the name of the SCHEMA in the upper half, which is divided from the lower half of the box by a horizontal line. The lower half of the symbol is empty.

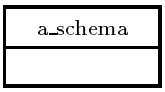


Figure D.9 – Symbol for a schema.

D.3 Relationship symbols

Definition symbols are connected by lines of various styles as shown in figure D.10.

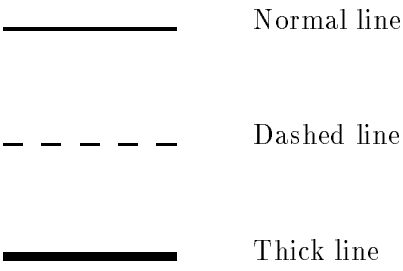


Figure D.10 – Relationship line styles

A relationship for an optional attribute of an entity data type is presented as a dashed line. A schema-schema reference is presented as a dashed line. An inheritance relationship (i.e., a subtype and supertype relationship) is presented as a thick line. All other relationships are presented as normal width solid lines.

Relationships are bidirectional, but one of the two directions is emphasized. If an entity A has an explicit attribute that is entity B, the emphasized direction is from A to B. In *EXPRESS-G*, the relationship is marked with an open circle in the emphasized direction, in this case, at the B end of the line. For an inheritance relationship, the emphasized direction is toward the subtype, i.e., the circle is at the subtype end of the line.

EXAMPLE 162 – Relationship directions are illustrated in figure D.11, which is an incomplete rendition of the *EXPRESS* code given in example 172 of annex H. The diagram consists of six

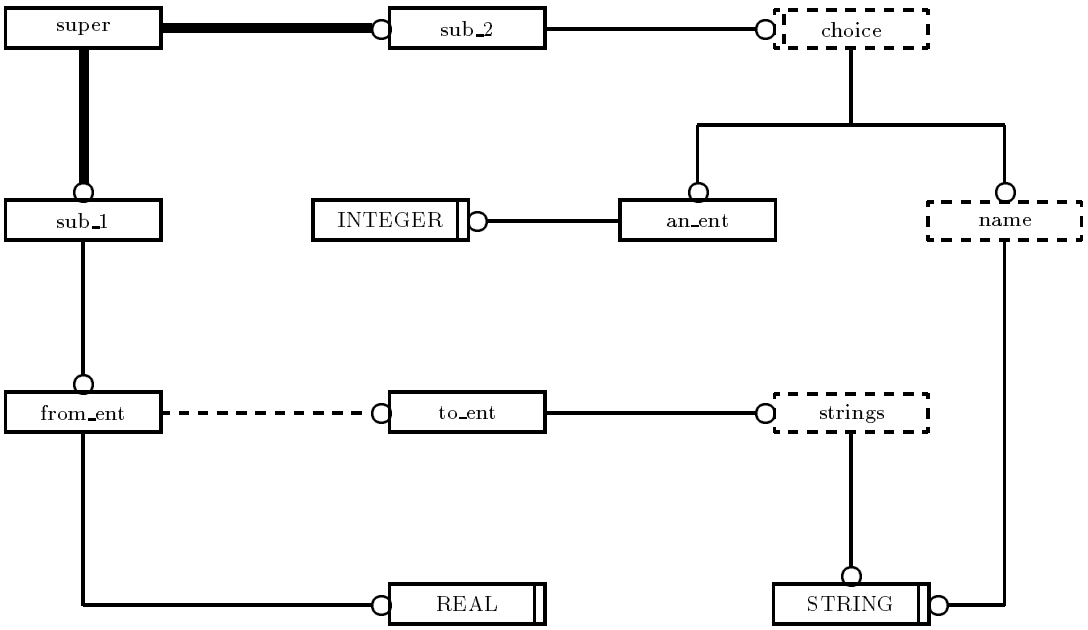


Figure D.11 – Partial entity level diagram illustrating relationship directions from example 172. (Page 1 of 1)

entity data types, three defined data types and several simple data types. Entity `super` has two subtypes, namely `sub_1` and `sub_2`. Entity `sub_2` has an attribute that is a select data type called `choice`, selecting between an entity data type named `an_ent` and the defined data type `name`. The entity data type `an_ent` has the integer data type as its attribute, while `name` is a string data type.

The entity data type `sub_1` has the entity data type `from_ent` as an attribute. `From_ent` has `to_ent` as an optional attribute and the real data type as a required attribute. In turn, the entity data type `to_ent` has a defined data type called `strings` as a required attribute, and `strings` is a list (not shown in diagram) of string data type.

NOTES

1 – Although the example diagrams show only straight relationship lines, lines may follow any path (e.g., they may be curved).

2 – It may not always be convenient to lay out a diagram without some of the relationship lines crossing each other. The means of distinguishing crossing points is left to the author of the diagram.

D.4 Composition symbols

Graphical representations can span more than one page. Each page is numbered. The symbols for achieving inter-page references are provided in figure D.12.

A schema may reference definitions from another schema. The symbols for inter-schema references are also shown in figure D.13.

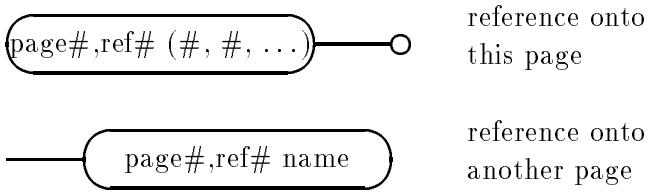


Figure D.12 – Composition symbols: page references

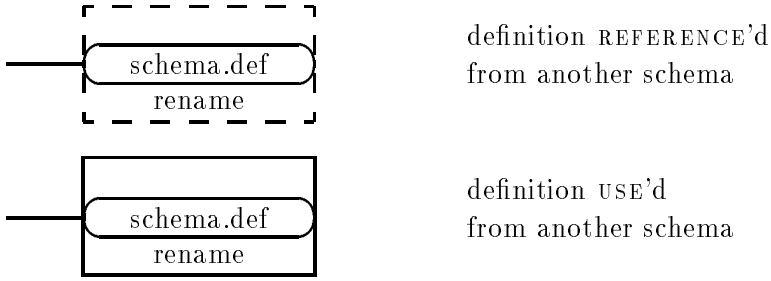


Figure D.13 – Composition symbols: inter-schema references

D.4.1 Page references

When there is a relationship between definitions on separate pages, the relationship line on the two pages is terminated by a rounded box. The rounded box contains a page number and a reference number, as shown in figure D.12. The page number is the number of the page where a referenced definition occurs. The reference number is used to distinguish between multiple references on a page. The composition symbol on the page where the reference originated contains the name of the referenced definition. The rounded reference box on the referenced page may contain a parenthesized list of the page numbers where references originated.

NOTE – The use of page referencing is shown in figure D.1 and figure D.2. The rounded box labelled `2,5` from the `person` indicates that the definition is to be found on page 2 of the diagram as reference 5. On page 2 of the diagram as shown in figure D.2, the rounded box symbol reference into `date` indicates that this definition is referenced from another definition on another page of the diagram. The number in parentheses indicates that the referencing item is to be found on page 1 of the diagram.

D.4.2 Inter-schema references

Inter-schema references are indicated by a rounded box enclosing the name of the definition qualified by the schema name, as shown in figure D.13.

Definitions accessed from another schema via an *EXPRESS* REFERENCE statement are enclosed by a dashed rectangular box. If the definition is renamed, the new name may be placed within the box below the rounded box.

Definitions accessed from another schema via an *EXPRESS* USE statement are enclosed by a solid rectangular box. If the definition is renamed, the new name may be placed within the box below the rounded box.

NOTE – The use of inter-schema references is shown in figure D.17.

D.5 Entity level diagrams

EXPRESS-G may be used to represent the definitions, and their relationships, within one schema. The components of this diagram consist of simple data types, defined data types, entity data types, and relationship symbols, together with role and cardinality information as appropriate, which represent the contents of a single schema.

D.5.1 Role names

In *EXPRESS* an attribute of an entity data type is named for the role of the data type being referenced when an instance participates in the relationship established by the attribute. The text string representing the role name may be placed on the relationship line connecting the entity data type symbol to its attribute symbol. These role names shall be consistent with the scope and visibility rules as defined in 10.

D.5.2 Cardinalities

The attributes of entity and defined data types can be represented by aggregation data types (i.e., LIST, SET, BAG, and ARRAY). In *EXPRESS-G*, an aggregation is indicated on the relationship line for the attribute, following the attribute name. Only the first letter of the aggregation data type (i.e., A, B, L or S) is used and the `OF` is omitted. If an aggregation is not specified, the cardinality is one for a required relationship and zero or one for an optional attribute.

NOTE – The *EXPRESS* given in example 172 is fully displayed using *EXPRESS-G* in figure D.14. The components of a SELECT type are not role-named.

D.5.3 Constraints

EXPRESS-G provides no methods for defining constraints, other than cardinalities. The fact that something is constrained within an *EXPRESS* data specification may be denoted by preceding the name of the thing with an asterisk (*) symbol. The following rules apply:

– if an entity is a parameter in an *EXPRESS* RULE, the name of the entity may be preceded by an asterisk;

– if an attribute of an entity is constrained by either a UNIQUE clause or a WHERE clause within the entity, the name of the attribute may be preceded by an asterisk;

– if a defined type is constrained by a WHERE clause, the name of the defined type may be preceded by an asterisk;

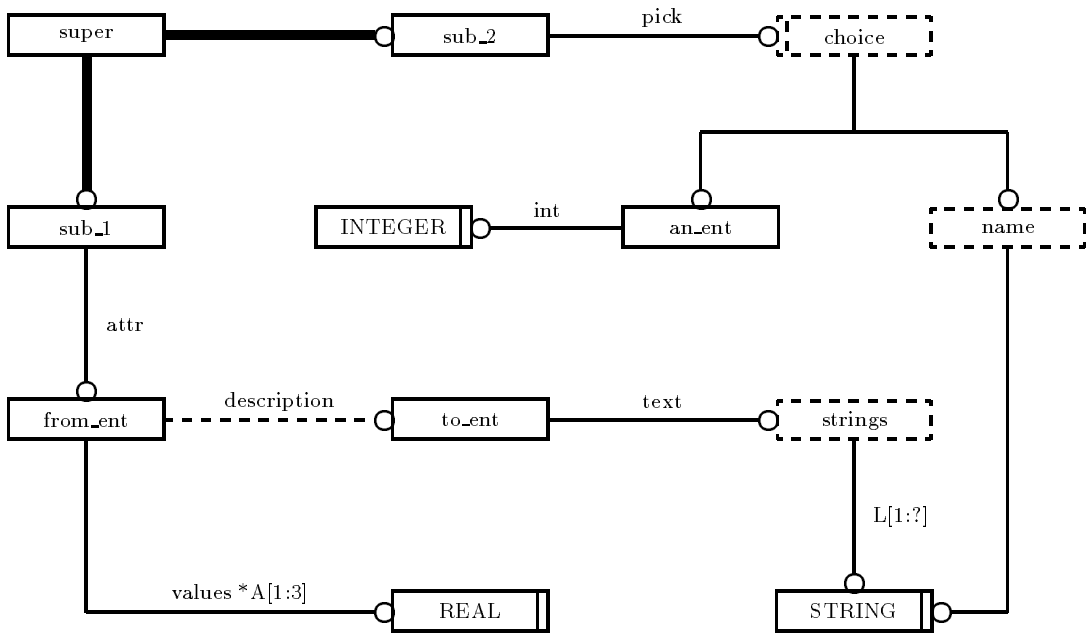


Figure D.14 – Complete entity level diagram of example 172. (Page 1 of 1)

– if an aggregation data type is constrained by a UNIQUE keyword, the first letter of the aggregation may be preceded by an asterisk.

D.5.4 Constructed and defined data types

A SELECT data type is represented by the select data type symbol (see figure D.4) plus a relationship and data type definition for each of the selectable items. No cardinality or role name is specified for the relationships.

An ENUMERATION data type is represented solely by its symbol (see figure D.4).

NOTE 1 – *EXPRESS-G* does not provide a mechanism for noting the enumeration items.

A defined data type is represented by the type definition symbol (see figure D.7) enclosing the name of the definition, the representation data type definition, and a relationship line from the defined data type definition to the representation data type definition. The cardinality of the representation may be placed on the relationship.

NOTE 2 – A defined data type representation can be seen in the `strings` type in figure D.14.

D.5.5 Entity data types

EXPRESS-G uses the solid box symbol (see figure D.8) for ENTITY definitions. The name of the entity data type is enclosed by the box.

In *EXPRESS-G* an ENTITY may:

– be part of an acyclic inheritance graph;

– have explicit attributes;

– have derived attributes;

– have inverse attributes.

Each explicit or derived attribute in an *EXPRESS* entity gives rise to a relationship in the corresponding *EXPRESS-G* diagram. The role name of the attribute may be placed on the relationship line, together with the cardinality which follows the attribute name. A derived attribute is distinguished from an explicit attribute by preceding the name of the attribute by the characters DER enclosed in parentheses, i.e (DER).

In the case where there is an inverse attribute defined for an attribute, the name and cardinality of the attribute is placed on the opposite side of the relationship line to the attribute name of which it is an inverse. The name is preceded by the characters INV enclosed in parentheses, i.e., (INV).

NOTES

1 – Typical entity level diagrams are shown in figure D.1 and figure D.14.

2 – The indication of domain rules applied to attributes can be seen on the roles husband and maiden-name in figure D.1.

3 – An example of entities constrained by Rules is shown for the `male` and `female` entities in figure D.1.

Subtype/supertype

The entities forming an inheritance graph are connected by thick solid lines. The circled end of the relationship line denotes the subtype end of the relationship. When a supertype is ABSTRACT the characters ABS, enclosed by parentheses, i.e., (ABS), precede the name of the entity within the entity symbol box.

EXPRESS-G provides a limited notation for indicating the logical structure of an inheritance graph. The ONEOF relation may be indicated by a branching relationship line from the supertype to each of its subtypes that are in a ONEOF relationship to each other, together with the digit 1 being placed at the branching junction.

NOTES

4 – Figure D.15 provides an *EXPRESS-G* diagram of example 173, showing `sub2` as being an Abstract Supertype.

5 – The diagram in figure D.15 shows that the entities `sub1`, `sub2` and `sub5` are subtypes of the supertype `super`. No implications can be made about the logical relationships between these subtypes (i.e., AND or ANDOR). An instance of `super` possibly has no subtypes because it is not ABSTRACT. The entities `sub3` and `sub4` are subtypes of the supertype `sub2`. The entities `sub3` and `sub4` are in a ONEOF relationship to each other.

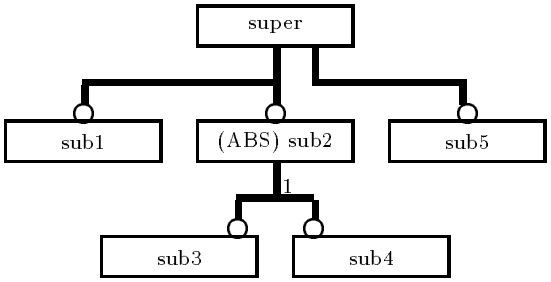


Figure D.15 – Complete entity level diagram of the inheritance graph from example 173. (Page 1 of 1)

EXPRESS permits the redeclaration of supertype attributes within a subtype, provided the redeclared attribute is a specialization of the supertype attribute type. In *EXPRESS-G* a redeclared attribute is represented in the same manner as its supertype attribute, but with the addition of the characters RT (redeclared type) enclosed in parentheses, i.e., (RT), before the name of the attribute.

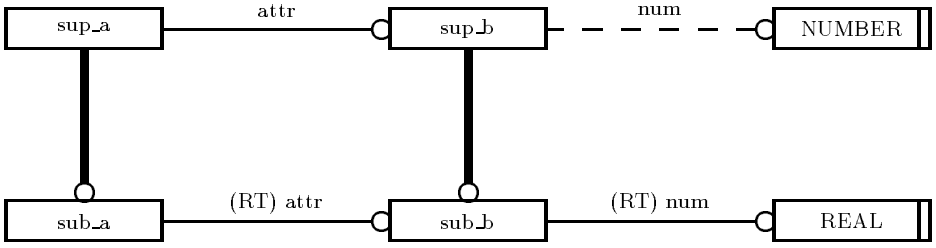


Figure D.16 – Complete entity level diagram of example 174 showing attribute redeclarations in subtypes. (Page 1 of 1)

NOTE 6 – Figure D.16 illustrates some of the forms of attribute redeclaration, as given in the *EXPRESS* example 174. Entity `sub_a` redeclares the attribute `attr` from its supertype to be a subtype of its supertype attribute. Entity `sup_b` has an optional attribute of type NUMBER. In its subtype, this is redeclared to be a required attribute of REAL type.

D.5.6 Inter-schema references

When a definition in the current schema refers to a definition in another schema, the inter-schema reference symbol is used and contains the qualified name of the definition.

NOTE – Figure D.17 shows a entity level diagram of a single schema. The *EXPRESS* source for this diagram is given as example 175. The complete diagram consists of two schemas, `top` and `geom` (see figure D.18), and some of the `top` schema entities have attributes that use definitions in the `geom` schema. Since a entity level diagram only consists of those things defined in a single schema, the representation of the `top` schema in this example requires the inter-schema references shown.

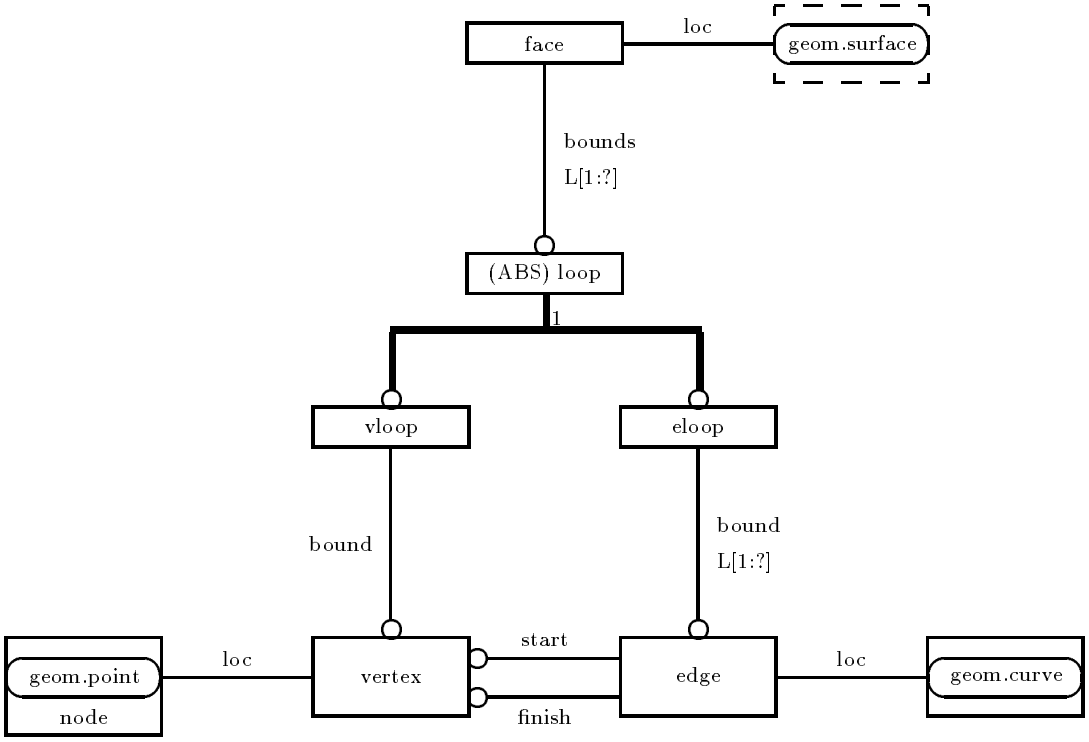


Figure D.17 – Complete entity level diagram of the top schema of example 175 illustrating inter-schema references. (Page 1 of 1).

D.6 Schema level diagrams

A schema level diagram consists of the representation of multiple schemas and their interfaces.

The contents of an *EXPRESS-G* schema level diagram are limited to the schemas comprising the diagram and the schema interfaces. The following are included:

- schemas that refer to another schema by means of USE;
- schemas that refer to another schema by means of REFERENCE;
- names of the things that are referenced or used.

The USE interface is presented by a normal width relationship line from the using schema to the used schema, with an open circle denoting the used schema. The REFERENCE interface is shown by a dashed relationship line from the referencing schema to the referenced schema, with an open circle denoting the referenced schema.

Definitions that are used or referenced may be shown as a list of names adjacent to the relevant relationship line, and connected to the relationship line by a line with an arrowhead adjacent to and pointing at the relationship line. The rename of a definition is indicated by following the original name of the definition by a greater than (>) sign and the rename.

NOTE 1 – A diagram with two schemas is shown in figure D.18. The `top` schema has an interface to the `geom` schema. In particular, the `top` schema references the `surface` and uses the `curve` and

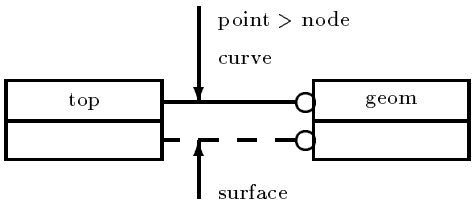


Figure D.18 – Complete schema level diagram of example 175. (Page 1 of 1)

`point` definitions from the `geom` schema. The `point` definition is renamed `node` in the `top` schema.

If a schema level diagram extends over more than one page and the schema interfaces cross the page boundaries, then the page referencing symbols are used.

NOTE 2 – Example 176 gives the *EXPRESS* source code for an abbreviated version of a schema level diagram. The *EXPRESS-G* schema diagram for this example is shown in figure D.19.

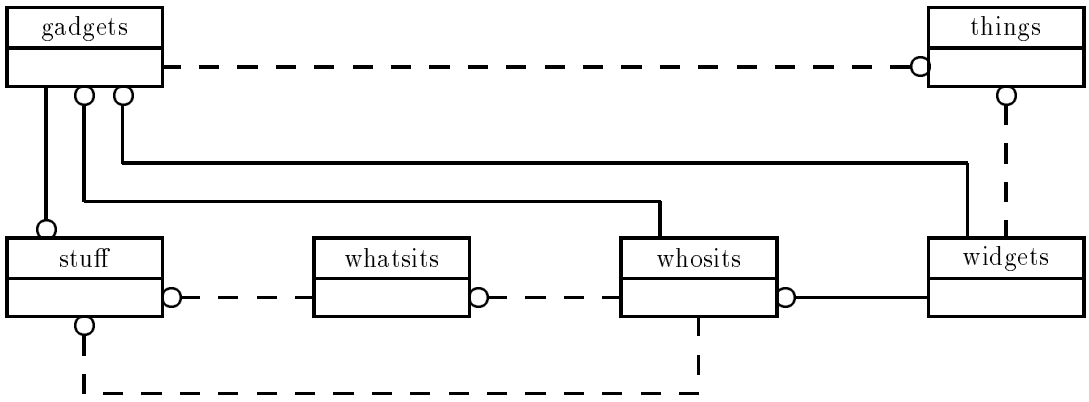


Figure D.19 – Complete schema level diagram of example 176. (Page 1 of 1)

D.7 Complete EXPRESS-G diagrams

In *EXPRESS-G* a complete diagram is one that, within the limits of the *EXPRESS-G* notation, accurately represents all the definitions, relationships and constraints using either an entity level or schema level diagram.

D.7.1 Complete entity level diagram

Diagrams that present a complete diagram of a single schema have contents defined by the following rules:

a) Each page shall have a title starting with the words: Complete entity level diagram of ...;

b) Each page is numbered in the form Page X of N, where N is the total number of pages forming the diagram, and X is the particular page number;

c) All entity data types, defined data types, and simple type symbols used within the single schema are displayed;

d) Schema symbols do not appear;

e) All relationships, attribute names, and cardinalities are displayed;

f) All attributes, including explicit, derived and inverse attributes are displayed;

g) All inheritance (i.e., subtype/supertype) relationships are displayed;

h) All ABSTRACT supertypes are marked;

i) All ONEOF subtype relations are marked;

j) All definitions used or referenced from another schema are presented by the rounded box symbols, together with the surrounding rectangular boxes of the appropriate style (solid for used definitions and dashed for referenced definitions);

k) Any rename is presented in the relevant inter-schema reference symbol;

l) All entities that are constrained by a RULE are marked with an asterisk (*);

m) All attributes that are constrained are marked with an asterisk (*);

n) All defined types that are constrained are marked with an asterisk (*);

o) All aggregation types that are constrained are marked with an asterisk (*).

All entity-entity relations not marked with an inverse attribute are interpreted to have a cardinality of zero or more. No logical structuring can be inferred from an unmarked subtype relation, except that it is not a ONEOF relation.

D.7.2 Complete schema level diagram

Diagrams that present a complete schema level diagram have contents defined by the following rules:

a) Each page is titled, with the title starting with the words: "Complete schema level diagram of ..."";

b) Each page is numbered in the form Page X of N, where N is the total number of pages forming the diagram, and X is the particular page number;

c) All schemas used are displayed;

d) Entity, type and simple symbols shall not be displayed;

e) All schema-schema relationships, USE and REFERENCE, are displayed;

f) The names of all definitions that are either used or referenced are attached to the relevant relationship line, together with any renames. If no names are attached to a relationship line, this is interpreted to mean that the entire schema is used or referenced.

NOTE – When developing models or displaying diagrams, it is useful to be able to display diagrams at varying levels of abstraction. For example, not all attributes are given on the diagram, or role names may not be shown. This is outside the scope of *EXPRESS-G*, but it is recommended that the level of abstraction be agreed and documented before development starts. Further, it is recommended that the diagram titles reflect the abstractions being used.

Annex E
(normative)
Protocol implementation conformance statement (PICS)

Is this implementation an *EXPRESS* language parser/verifier? If so, answer the questions provided in E.1.

Is this implementation an *EXPRESS-G* editing tool? If so, answer the questions provided in E.2.

E.1 EXPRESS language parser

For which level is support claimed:

☐ Level 1 – Reference checking;
☐ Level 2 – Type checking;
☐ Level 3 – Value checking;
☐ Level 4 – Complete checking.

(Note: In order to claim support for a given level, all lower levels must also be supported.)

What is the maximum integer value [integer_literal]?: .....:

What is the maximum real precision [real_literal]?: .....:

What is the maximum real exponent [real_literal]?: .....:

What is the maximum string width (characters) [simple_string_literal]?: .....:

What is the maximum string width (octets) [encoded_string_literal]?: .....:

What is the maximum binary width (bits) [binary_literal]?: .....:

Do you have a limit on the number of unique identifiers which are declared? If so, what is your limit?: .....:

Do you have a limit on the number of characters used as an identifier? If so, what is your limit?: .....:

Do you have a limit on the scope nesting depth? If so, what is your limit?: .....:

Do you implement the concept of multiple name scopes in which schema names may occur? If so, what are these scopes called?: .....:

How do you represent the standard constant '?' [built_in_constant]?: .....:

E.2 EXPRESS-G editing tool

For which level is support claimed:

☐ Level 1 – Symbol checking;
☐ Level 2 – Complete checking.

(Note: In order to claim support for a given level, all lower levels must also be supported.)

Do you have a limit on the number of unique identifiers which are declared? If so, what is your limit?: .....:

Do you have a limit on the number of characters used as an identifier? If so, what is your limit?: .....:

Do you have a limit on the number of symbols per page of the model? If so, what is your limit?: .....:

Do you have a limit on the number of pages available to a model? If so, what is your limit?: .....:

Do you implement the concept of multiple name scopes in which schema names may occur? If so, what are these scopes called?: .....:

Do you implement the full form of constructed data type representation, the abbreviated form or both forms?: .....:

Annex F
(normative)
Information object registration

In order to provide for unambiguous identification of an information object in an open system, the object identifier

{ iso standard 10303 part(11) version(1) }

is assigned to this part of ISO 10303. The meaning of this value is defined in ISO 8824-1, and is described in ISO 10303-1.

Annex G
(informative)
Relationships

G.1 Relationships via attributes

In *EXPRESS* the declaration in an entity data type of an attribute whose domain is another data type explicitly establishes a relationship between these two data types. This relationship is referred to as a simple relationship, and relates an instance of the declaring entity to one instance of the representing data type.

To characterize relationships established by aggregate valued attributes, define the fundamental base type of a data type as the non-aggregation data type given by:

– the fundamental base type of a non-aggregation data type is the data type itself;

– the fundamental base type of an aggregation data type is the fundamental base type of its base type.

When the fundamental base type of an attribute `A` is `T`, we say that `A` is founded on `T`.

Then, the declaration in an entity data type of an attribute whose domain is an aggregation data type founded on a fundamental base type establishes two sorts of relationships:

– A collective relationship between the declaring entity and the aggregation data type. This relates an instance of the declaring entity to a collection of instances of the fundamental base type.

– A distributive relationship between the declaring entity and the fundamental base type. This relates an instance of the declaring entity to one or more instances of the fundamental base type individually.

NOTE – This approach differs from some other modelling languages. For example in an Entity-Relationship (ER) model, entities and relationships are modelled by different constructs.

Both simple and distributive relationships are directed from the declaring entity to some other data type. It is useful to discuss the cardinality of these relationships (from the point of view of the declaring entity). If this cardinality is $m : n$ $(0 \leq m \leq n)$, every instance of the declaring entity is associated with at least m and at most n instances of the target data type. If n is indeterminate (`?`), there is no upper limit on the number of instances of the target data type with which an instance of the declaring entity may be associated.

It is useful to discuss an inverse relationship, which is the reverse direction of a simple or distributive relationship. This relationship always implicitly exists and by default has a cardinality of $0 : ?$. It may be explicitly named and optionally constrained by an INVERSE attribute declaration in the representing data type if the representing data type is an entity data type.

EXAMPLE 163 – In this example there is a simple relationship between the entity data types `first` and `second` in which `second` plays the role `ref`. The cardinality of this relationship with respect to `first` in this case is $1 : 1$ (i.e. every instance of `first` is related to exactly one instance of `second`).

The cardinality of this relationship with respect to `second` is 0 : ?, or unconstrained (i.e. one instance of `second` may be related to zero or more instances of `first`); this is the default cardinality of the inverse relationship.

```
ENTITY first;
  ref   : second;
  fattr : STRING;
END_ENTITY;

ENTITY second;
  sattr : STRING;
END_ENTITY;
```

If an entity data type `E` has a relationship to a data type `T` established by an attribute `A`, this relationship can be diagrammed as:

$$\texttt{E.A} \xrightarrow{\{m:n\}\quad\{p:q\}} \texttt{T}$$

with $0 \leq m \leq n$ and $0 \leq p \leq q$. Here, $m : n$ is the cardinality of the forward relationship from `E` to `T`, while $p : q$ is the cardinality of the inverse relationship from `T` to `E`.

The three sorts of relationships and their associated cardinalities are more formally described below.

G.1.1 Simple relationship

A simple relationship is the relationship established by an attribute whose representation is another entity data type. This relationship is established between the two entity data types involved.

A simple relationship always exists between an instance of the declaring entity and at most one instance of the representing entity. Using the diagramming scheme above, this can be shown as:

$$\texttt{E.A} \xrightarrow{\{m:1\}\quad\{p:q\}} \texttt{T}$$

with $0 \leq m \leq 1$ and $0 \leq p \leq q$.

This means that for every instance of `E`, the role `A` is played either by no instance or by exactly one instance of type `T`. For every instance of `T`, there have to be between p and q instances of `E` in which this instance of `T` plays the role `A`.

The following cases for the values of p and q are meaningful classes of constraints on the simple relationship between `E` and `T`:

– if $q = 1$, there is a constraint that an instance of `T` may not play the role `A` in more than one instance of `E`;

– if $1 \leq p$, there is an existence constraint on `T`. That is, for every instance of `T` there must exist at least p (but no more than q) instances of `E` using this instance of `T` in the role `A`.

Several different *EXPRESS* constructs are used to constrain the cardinality of a simple relationship and of its inverse relationship:

– the case $m = 0$ is provided for by declaring the attribute `A` to be OPTIONAL. If `A` is not declared OPTIONAL, $m = 1$;

– the case $q = 1$ is provided for by declaring a simple inverse attribute, or by attaching a uniqueness rule to `E.A`, which requires that each role `A` in the population of `E`'s uses a different instance, therefore an instance of `T` can be used by at most one `E.A`;

– other constraints on the cardinality of the inverse relationship are expressed by declaring an INVERSE attribute in `T`, as `INVERSE I : SET [p:q] OF E FOR A`. The case where $p = q = 1$ can be abbreviated as `INVERSE I : E FOR A`.

Some examples of simple relationships and the associated cardinality constraints follow:

EXAMPLES

164 – `CIRCLE.CENTRE` $\xrightarrow{\{1:1\}\quad\{0:?\}}$ `POINT`

Every `CIRCLE` has exactly one `POINT` playing the role of `CENTRE`. Every `POINT` may play the role of `CENTRE` in any number of `CIRCLE`s (including none). This could be declared by:

```
ENTITY point;
  ...
END_ENTITY;

ENTITY circle;
  centre : point;
  ...
END_ENTITY;
```

165 – `PRODUCT_VERSION.BASE_PRODUCT` $\xrightarrow{\{1:1\}\quad\{1:?\}}$ `PRODUCT`

Every `PRODUCT_VERSION` has exactly one `PRODUCT` playing the role of `BASE_PRODUCT`. A `PRODUCT` may play the role `BASE_PRODUCT` in any number of `PRODUCT_VERSION`s, but must play this role in least one (existence dependence). This could be declared by:

```
ENTITY product_version;
  base_product : product; ...
END_ENTITY;

ENTITY product;
  ...
INVERSE
  versions : SET [1:?] OF product_version FOR base_product;
  ...
END_ENTITY;
```

166 – `PERSON.LUNCH` $\xrightarrow{\{0:1\}\quad\{0:?\}}$ `MEAL`

Each `PERSON` may have a `MEAL` playing the role of `LUNCH`. A `MEAL` may play the role `LUNCH` for any number of `PERSON` (assuming it is large enough!). This could be declared by:

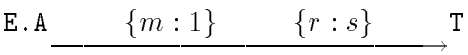
```
ENTITY person;
  lunch : OPTIONAL meal;
  ...
END_ENTITY;

ENTITY meal;
  calories : energy_measure;
  amount   : weight_measure;
  ...
END_ENTITY;
```

G.1.2 Collective relationship

An aggregate-valued attribute of an entity data type establishes a collective relationship between the entity data type and the aggregation data type used to represent the attribute.

NOTE – The collective relationship does not involve the entity instances of which the attribute aggregate values are ultimately composed. These instances participate instead in the distributive relationship (see G.1.3).

A collective relationship is similar to a simple relationship in the non-aggregate case. A collective relationship always exists between an instance of the declaring entity and at most one instance of the representing aggregation data type. As in the simple relationship case, this can be diagrammed as:



with $0 \leq m \leq 1$ and $0 \leq r \leq s$.

The following cases for the values of r and s are meaningful classes of constraints on the collective relationship between E and T:

- if $s = 1$, there is a uniqueness constraint on the collective value of attribute A;
- if $1 \leq r$, there is an existence constraint on T.

As with the simple relationship, m is controlled by declaring A to be OPTIONAL ($m = 0$). The uniqueness constraint, where $s = 1$, can be captured, as in the simple relationship case, by writing a uniqueness rule on A in the declaration of E. Otherwise, r and s cannot be constrained.

Some examples of collective relationships and the associated cardinality constraints follow:

EXAMPLES

167 – POLY_CURVE.COEF {1 : 1} {0 : ?} LIST [1:?] OF REAL

Every POLY_CURVE has a list of REAL numbers playing the role of COEF. Any LIST [1:?] OF REAL may play the role of COEF in any number of POLY_CURVEs (including none). This could be declared by:

```
ENTITY poly_curve;
  coef : LIST [1:?] OF REAL;
  ...
```

```
END_ENTITY;
```

168 – LOOP.EDGES $\xrightarrow{\{0:1\}\quad\{0:1\}}$ LIST [1:?] OF EDGE

Every LOOP may have a list of EDGE playing the role of EDGES. Every LIST [1:?] OF EDGE may play the role EDGES for at most one LOOP instance. This could be declared by:

```
ENTITY loop;
  edges : OPTIONAL LIST [1:?] OF edge;
  ...
UNIQUE
  un1 : edges;
END_ENTITY;

ENTITY edge;
     ...
END_ENTITY;
```

G.1.3 Distributive relationship

In addition to the collective relationship discussed above, an aggregate-valued attribute establishes a distributive relationship between the entity data type and the fundamental base type of the aggregation data type used to represent the attribute.

A distributive relationship relates an instance of the declaring entity individually to any number of instances of the representing fundamental base type. The cardinality of this relationship is constrained by the cardinality of the aggregation data type(s) used to represent the attribute. Writing FUND(T) for the fundamental base type of the attribute type, the distributive relationship can be diagrammed as:

E.A $\xrightarrow{\{k:l\}\quad\{p:q\}}$ FUND(T)

with $0 \leq k \leq l$ and $0 \leq p \leq q$.

This means that for every instance of E the attribute A consists of between k and l instances of FUND(T). An instance of FUND(T) may appear in between p and q instances of E in the role A.

The following cases for the values of p and q are meaningful classes of constraints on the distributive relationship between E and FUND(T):

– if $q = 1$, there is a constraint that an instance of FUND(T) may not appear in the role A in more than one instance of E;

– if $1 \leq p$, there is an existence constraint on FUND(T). That is, for every instance of FUND(T) there must exist at least p (but no more than q) instances of E which contain the instance of FUND(T) in the role A.

The following *EXPRESS* constructs are used to constrain the cardinality of a distributive relationship and its inverse relationship.

– the values of k and l are captured by the bound specifications of the aggregation data types used to represent `A`. In the simplest case, the data type of the attribute will simply be `SET [k:l] OF FUND(T)` (or a similar BAG or LIST data type);

NOTE 1 – In this approach to relationships, there is no distinction between one-dimensional and multi-dimensional aggregate values.

– the case $q = 1$ for a distributive relationship cannot be captured by attaching a uniqueness rule to `E.A`. Instead, an INVERSE attribute must be declared and constrained in `FUND(T)`, as `INVERSE I : E FOR A;`

– other constraints on the cardinality of the inverse relationship are expressed by declaring an INVERSE attribute in `FUND(T)`, as `INVERSE I : SET [p:q] OF E FOR A`. The case where $p = q = 1$ can be abbreviated as `INVERSE I : E FOR A`.

Some examples of distributed relationships and the associated cardinality constraints follow:

EXAMPLES

169 – Contrast this with example 167.

`POLY_CURVE.COEF` $\xrightarrow{\{1:?\}\quad\{0:?\}}$ `REAL`

A `POLY_CURVE` has at least one REAL number playing the role of `COEF`. A particular REAL number may be used within the `COEF` attribute of an unlimited number of `POLY_CURVE`s (including none). This could be declared by:

```
ENTITY poly_curve;
  coef : LIST [1:?] OF REAL;
  ...
END_ENTITY;
```

170 – Compare this with example 168.

`LOOP.EDGES` $\xrightarrow{\{0:?\}\quad\{2:2\}}$ `EDGE`

A `LOOP` may consist of any number of `EDGE`s (including none). An `EDGE` has to be used for exactly two different `LOOP`s. This could be declared by:

```
ENTITY loop;
  edges : OPTIONAL LIST [1:?] OF edge;
  ...
UNIQUE
  un1 : edges;
END_ENTITY;

ENTITY edge;
  ...
INVERSE
```

```
  loops : SET [2:2] OF loop FOR edges;
  ...
END_ENTITY;
```

G.1.4 Inverse attribute

Every relationship established by an attribute has an implicit inverse relationship. By default, this relationship is effectively ignored, i.e., it cannot be referenced, and its cardinality is unconstrained. A uniqueness rule on the attribute declaring a simple relationship effectively constrains the cardinality of the inverse relationship. *EXPRESS* does provide constructs which allow inverse relationships to be named and constrained. These constructs have been partially described in relation to the other classes of relationships, and are summarized here.

An inverse relationship is given an identifier by the declaration of an INVERSE attribute. The type of the INVERSE attribute can constrain the cardinality of this inverse relationship.

In the following discussion of particular *EXPRESS* constructs and their effects on inverse cardinality, an entity `E` is assumed to declare an attribute `A` of data type `T`. If `T` is an aggregation data type, its fundamental base type is `FUND(T)`. The representing entity (`FUND(T)` or `T`, as appropriate) is denoted `R`. The inverse relationship is assumed to be declared by an INVERSE attribute `I` within `R`.

– if `A` is a non-aggregate attribute with a uniqueness rule associated with it, the simple relationship is constrained such that in the population of `E` each `A` is unique. Therefore an instance of `T` can play the role `A` in no more than one instance of `E`, i.e., $q = 1$. This is equivalent to `INVERSE I : SET [0:1] OF E FOR A;`

– if `A` is an aggregate attribute with a uniqueness rule associated with it, the collective relationship is constrained such that $s = 1$. That is, an instance of `T` (which is an aggregate) can play the role `A` in no more than one instance of `E`. There is no constraint on the distributive relationship: an instance of `R` may play the role `A` in any number of instances of `E`;

– if `I` is declared as `INVERSE I : BAG [p:q] OF E FOR A`, the cardinality of the inverse direction of the simple or distributive relationship is constrained according to the values of `p` and `q`. That is, an instance of `R` may play the role `A` in between `p` and `q` instances of `E`. Since a BAG admits instance equal elements, a particular instance of `R` may play this role more than once in a particular instance of `E`. This is only meaningful if `T` is an aggregation data type which itself admits duplicate elements;

– if `I` is declared as `INVERSE I : SET [p:q] OF E FOR A`, the cardinality of the inverse direction of the simple or distributive relationship is constrained according to the values of `p` and `q`. That is, an instance of the `R` may play the role `A` in between `p` and `q` instances of `E`. Since a SET does not admit duplicate elements, a particular instance of `R` may not play this role more than once in particular instance of `E`.

– if `I` is declared as `INVERSE I : E FOR A`, the effect is the same as if if had been declared as a `SET [1:1] OF E`. That is, every instance of `R` must play the role `A` in exactly one instance of `E`;

– any declaration of I which is either a BAG or SET with $p \geq 1$ or neither a BAG nor a SET;

establishes an existence constraint on R: it requires that every instance of R play the role A in at least one instance of E.

G.2 Subtype/supertype relationships

The subtype declaration within an entity specifies a relationship between the subtype entity and the specified supertype entities.

Given a supertype entity P which has the subtype C, the relationship can be diagrammed as:

$$\mathtt{P} \xrightarrow{\{n:1\} \quad \{1:1\}} \mathtt{C}$$

with $0 \leq n \leq 1$. This means that for every instance of P, there is either zero or one instances of C. For every instance of C, there is one instance of P.

In the case when P is an ABSTRACT supertype, the relationship is:

$$\mathtt{P} \xrightarrow{\{1:1\} \quad \{1:1\}} \mathtt{C}$$

This means that for every instance of P, there is one instance of C. For every instance of C, there is one instance of P.

Annex H
(informative)
EXPRESS models for EXPRESS-G illustrative examples

This annex provides the *EXPRESS* rendition of several examples that have been used to illustrate *EXPRESS-G* modelling.

No claim is made that any of these models are either realistic or "good". In particular, these example models have no relationship whatsoever with any models in any other part of this International Standard.

H.1 Example single schema model

The model in example 171 basically says that a person must be either a male or a female. Every person has some defining characteristics, such as first and last names, date of birth, type of hair, and may also have zero or more children (which are, of course, also people). A male may be married to a female, in which case the female has an inverse relationship to the male.

The `age` of a person is a derived attribute that is calculated through the function `years` which determines the number of years between the date input as a parameter and the current date.

A `person` has an inverse attribute which relates people who are children to their parents. The lower bound of this inverse attribute is 0 to ensure we don't have to supply the whole family tree.

NOTE – If `parents` was a required explicit attribute and `children` was an inverse attribute, the family tree would have to extend backwards in time.

EXAMPLE 171 – A single schema EXPRESS model.

```
SCHEMA example;

TYPE date = ARRAY [1:3] OF INTEGER;
END_TYPE;

TYPE hair_type = ENUMERATION OF
     (blonde,
      brown,
      black,
      red,
      white,
      bald);
END_TYPE;

ENTITY person
  ABSTRACT SUPERTYPE OF (ONEOF(female, male));
  first_name : STRING;
  last_name  : STRING;
  nickname   : OPTIONAL STRING;
  birth_date : date;
```

```
  children   : SET [0:?] OF person;
  hair       : hair_type;
DERIVE
  age : INTEGER := years(birth_date);
INVERSE
  parents : SET [0:2] OF person FOR children;
END_ENTITY;

ENTITY female
  SUBTYPE OF (person);
INVERSE
  husband     : SET [0:1] OF male FOR wife; -- husband is optional !
END_ENTITY;

ENTITY male
  SUBTYPE OF (person);
  wife : OPTIONAL female;
END_ENTITY;

FUNCTION years(past : date): INTEGER;
  (* This function calculates the number of years
     between the past date and the current date *)
END_FUNCTION;

END_SCHEMA;
```

H.2 Relationship sampler

Example 172 is a simple model for the purposes of indicating some of the declarations and relationships to be found in an *EXPRESS* model. The model contains supertype entities, subtype entities, and entities that are neither of these. Also included are two defined data types, a select type and some simple types.

EXAMPLE 172 – A simple EXPRESS entity and type relationship model.

```
SCHEMA etr;

ENTITY super;
END_ENTITY;

ENTITY sub_1
  SUBTYPE OF (super);
  attr : from_ent;
END_ENTITY;

ENTITY sub_2
  SUBTYPE OF (super);
  pick : choice;
END_ENTITY;
```

```
ENTITY an_ent;
  int : INTEGER;
END_ENTITY;

ENTITY from_ent;
  description : OPTIONAL to_ent;
  values      : ARRAY [1:3] OF UNIQUE REAL;
END_ENTITY;

ENTITY to_ent;
  text : strings;
END_ENTITY;

TYPE choice = SELECT
   (an_ent,
    name);
END_TYPE;

TYPE name = STRING;
END_TYPE;

TYPE strings = LIST [1:?] OF STRING;
END_TYPE;

END_SCHEMA;
```

H.3 Simple subtype/supertype tree

EXPRESS enables very complex subtype/supertype trees (and networks) to be defined. The tree shown in example 173 is relatively simple.

EXAMPLE 173 – Subtype/supertype tree in EXPRESS

```
SCHEMA simple_trees;

ENTITY super;
END_ENTITY;

ENTITY sub1
  SUBTYPE OF (super);
END_ENTITY;

ENTITY sub2
  ABSTRACT SUPERTYPE OF (ONEOF(sub3,
                               sub4))
  SUBTYPE OF (super);
END_ENTITY;

ENTITY sub3
  SUBTYPE OF (sub2);
END_ENTITY;
```

```
ENTITY sub4
  SUBTYPE OF (sub2);
END_ENTITY;

ENTITY sub5
  SUBTYPE OF (super);
END_ENTITY;

END_SCHEMA;
```

H.4 Attribute redeclaration

EXPRESS permits the redeclaration of inherited attributes, provided the new attribute types are compatible. Example 174 shows some of the permissable forms of redeclaration:

– the redeclared attribute type is a subtype of the inherited type;

– the redeclared attribute type is a compatible simple type;

– the value of the redeclared attribute is required whereas the inherited value was optional.

EXAMPLE 174 – EXPRESS attribute redeclaration.

```
ENTITY sup_a;
  attr : sup_b;
END_ENTITY;

ENTITY sub_a
  SUBTYPE OF (sup_a);
  SELF\sup_a.attr : sub_b;
END_ENTITY;

ENTITY sup_b;
  num : OPTIONAL NUMBER;
END_ENTITY;

ENTITY sub_b
  SUBTYPE OF (sup_b);
  SELF\sup_b.num : REAL;
END_ENTITY;
```

H.5 Multi-schema models

EXPRESS models consist of at least one schema. Example 175 shows a model that consists of two schemas.

EXAMPLE 175 – A two schema EXPRESS model.

```
SCHEMA geom;
```

```
  ENTITY lcs;
  END_ENTITY;

  ENTITY surface;
  END_ENTITY;

  ENTITY curve;
  END_ENTITY;

  ENTITY point;
  END_ENTITY;

END_SCHEMA; -- geom

SCHEMA top;
  USE FROM geom
      (curve,
       point AS node);
  REFERENCE FROM geom
      (surface);

  ENTITY face;
    bounds : LIST [1:?] OF loop;
    loc    : surface;
  END_ENTITY;

  ENTITY loop
    ABSTRACT SUPERTYPE OF
             (ONEOF(eloop, vloop));
  END_ENTITY;

  ENTITY eloop
    SUBTYPE OF (loop);
    bound : LIST [1:?] OF edge;
  END_ENTITY;

  ENTITY vloop
    SUBTYPE OF (loop);
    bound : vertex;
  END_ENTITY;

  ENTITY edge;
    start : vertex;
    end   : vertex;
    loc   : curve;
  END_ENTITY;

  ENTITY vertex;
    loc : node;
  END_ENTITY;
```

```
END_SCHEMA; -- top
```

A more complicated set of schemas is given in example 176. It is to be understood in this case that within each of the declared schemas, there are entity, type and other definitions, although these are not shown in order to conserve space.

EXAMPLE 176 – EXPRESS multi-schema model.

```
SCHEMA stuff;
END_SCHEMA;

SCHEMA whatsits;
  REFERENCE FROM stuff;
END_SCHEMA;

SCHEMA widgets;
  USE FROM whosits;
  USE FROM gadgets;
  REFERENCE FROM things;
END_SCHEMA;

SCHEMA things;
END_SCHEMA;

SCHEMA gadgets;
  USE FROM stuff;
  REFERENCE FROM things;
END_SCHEMA;

SCHEMA whosits;
  REFERENCE FROM stuff;
  REFERENCE FROM whatsits;
END_SCHEMA;
```

Annex J
(informative)
Bibliography

1. ISO TR/9007:1987, *Information processing systems - Concepts and terminology for the conceptual schema and the information base.*

2. KAMADA, T. and KAWAI, S.; *"A General Framework for Visualizing Abstract Objects and Relations"*, ACM Transactions on Graphics, January 1991, vol. 10, no. 1, p. 1-39.

3. WIRTH, N.; *"What can we do about the unnecessary diversity of notation for syntactic definitions?"*, Communications of the ACM, November 1977, vol. 20, no. 11, p. 822.

Index