INTERNATIONAL STANDARD

ISO 10303-11

Second edition
2004-11-01

Industrial automation systems and integration — Product data representation and exchange —

Part 11: Description methods: The EXPRESS language reference manual

Systèmes d'automatisation industrielle et intégration — Représentation et échange de données de produits —

Partie 11: Méthodes de description: Manuel de référence du langage EXPRESS

Reference number
ISO 10303-11:2004(E)

© ISO 2004

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS

Not for Resale

PDF disclaimer

This PDF file may contain embedded typefaces. In accordance with Adobe's licensing policy, this file may be printed or viewed but shall not be edited unless the typefaces which are embedded are licensed to and installed on the computer performing the editing. In downloading this file, parties accept therein the responsibility of not infringing Adobe's licensing policy. The ISO Central Secretariat accepts no liability in this area.

Adobe is a trademark of Adobe Systems Incorporated.

Details of the software products used to create this PDF file can be found in the General Info relative to the file; the PDF-creation parameters were optimized for printing. Every care has been taken to ensure that the file is suitable for use by ISO member bodies. In the unlikely event that a problem relating to it is found, please inform the Central Secretariat at the address given below.

© ISO 2004

All rights reserved. Unless otherwise specified, no part of this publication may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying and microfilm, without permission in writing from either ISO at the address below or ISO's member body in the country of the requester.

ISO copyright office
Case postale 56 • CH-1211 Geneva 20
Tel. + 41 22 749 01 11
Fax + 41 22 749 09 47
E-mail copyright@iso.org
Web www.iso.org

Published in Switzerland

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Contents

Page

0 Introduction . . . xii
0.1 General . . . xii
0.2 Language overview . . . xii

1 Scope . . . 1

2 Normative references . . . 2

3 Terms and definitions . . . 2
3.1 Terms defined in ISO 10303–1 . . . 2
3.2 Terms defined in ISO/IEC 10646 . . . 2
3.3 Other terms and definitions . . . 2

4 Conformance requirements . . . 5
4.1 Formal specifications written in EXPRESS . . . 5
4.1.1 Lexical language . . . 5
4.1.2 Graphical form . . . 6
4.2 Implementations of EXPRESS . . . 6
4.2.1 EXPRESS language parser . . . 6
4.2.2 Graphical editing tool . . . 6

5 Fundamental principles . . . 7

6 Language specification syntax . . . 8
6.1 The syntax of the specification . . . 8
6.2 Special character notation . . . 9

7 Basic language elements . . . 9
7.1 Character set . . . 10
7.1.1 Digits . . . 10
7.1.2 Letters . . . 10
7.1.3 Special characters . . . 11
7.1.4 Underscore . . . 11
7.1.5 Whitespace . . . 11
7.1.6 Remarks . . . 11
7.2 Reserved words . . . 14
7.2.1 Keywords . . . 14
7.2.2 Reserved words which are operators . . . 14
7.2.3 Built-in constants . . . 15
7.2.4 Built-in functions . . . 15
7.2.5 Built-in procedures . . . 15
7.3 Symbols . . . 15
7.4 Identifiers . . . 15
7.5 Literals . . . 16
7.5.1 Binary literal . . . 16
7.5.2 Integer literal . . . 17
7.5.3 Real literal . . . 17
7.5.4 String literal . . . 18
7.5.5 Logical literal . . . 19

8 Data types . . . 19
8.1 Simple data types . . . 20

©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

8.1.1 Number data type . . . 20
8.1.2 Real data type . . . 20
8.1.3 Integer data type . . . 21
8.1.4 Logical data type . . . 21
8.1.5 Boolean data type . . . 21
8.1.6 String data type . . . 21
8.1.7 Binary data type . . . 22
8.2 Aggregation data types . . . 23
8.2.1 Array data type . . . 24
8.2.2 List data type . . . 25
8.2.3 Bag data type . . . 26
8.2.4 Set data type . . . 26
8.2.5 Value uniqueness on aggregates . . . 27
8.3 Named data types . . . 28
8.3.1 Entity data type . . . 28
8.3.2 Defined data type . . . 29
8.4 Constructed data types . . . 29
8.4.1 Enumeration data type . . . 30
8.4.2 Select data type . . . 33
8.5 Generalized data types . . . 35
8.6 Data type usage classification . . . 35
8.6.1 Instantiable data types . . . 37
8.6.2 Parameter data types . . . 37
8.6.3 Underlying data types . . . 37

9 Declarations . . . 38
9.1 Type declaration . . . 38
9.2 Entity declaration . . . 40
9.2.1 Attributes . . . 41
9.2.2 Local rules . . . 45
9.2.3 Subtypes and supertypes . . . 48
9.2.4 Abstract entity data type . . . 54
9.2.5 Subtype/supertype constraints . . . 55
9.2.6 Implicit declarations . . . 60
9.2.7 Specialization . . . 61
9.3 Schema . . . 62
9.4 Constant . . . 63
9.5 Algorithms . . . 64
9.5.1 Function . . . 64
9.5.2 Procedure . . . 65
9.5.3 Parameters . . . 65
9.5.4 Local variables . . . 71
9.6 Rule . . . 72
9.7 Subtype constraints . . . 74
9.7.1 Abstract supertype constraint . . . 75
9.7.2 Total coverage subtypes . . . 75
9.7.3 Overlapping subtypes and their specification . . . 76

10 Scope and visibility . . . 78
10.1 Scope rules . . . 78
10.2 Visibility rules . . . 79
10.3 Explicit item rules . . . 81
10.3.1 Alias statement . . . 81

 ©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

10.3.2 Attribute . . . 81
10.3.3 Constant . . . 81
10.3.4 Enumeration item . . . 81
10.3.5 Entity . . . 81
10.3.6 Function . . . 82
10.3.7 Parameter . . . 83
10.3.8 Procedure . . . 83
10.3.9 Query expression . . . 83
10.3.10 Repeat statement . . . 84
10.3.11 Rule . . . 84
10.3.12 Rule label . . . 85
10.3.13 Schema . . . 85
10.3.14 Subtype constraint . . . 86
10.3.15 Type . . . 86
10.3.16 Type label . . . 86
10.3.17 Variable . . . 86

11 Interface specification . . . 86
11.1 Use interface specification . . . 87
11.2 Reference interface specification . . . 87
11.3 The interaction of use and reference . . . 88
11.4 Implicit interfaces . . . 89
11.4.1 Constant interfaces . . . 89
11.4.2 Defined data type interfaces . . . 90
11.4.3 Entity data type interfaces . . . 90
11.4.4 Function interfaces . . . 91
11.4.5 Procedure interfaces . . . 91
11.4.6 Rule interfaces . . . 91
11.4.7 Subtype constraint interfaces . . . 91

12 Expression . . . 92
12.1 Arithmetic operators . . . 93
12.2 Relational operators . . . 94
12.2.1 Value comparison operators . . . 95
12.2.2 Instance comparison operators . . . 99
12.2.3 Membership operator . . . 101
12.2.4 Interval expressions . . . 102
12.2.5 Like operator . . . 102
12.3 Binary operators . . . 103
12.3.1 Binary indexing . . . 103
12.3.2 Binary concatenation operator . . . 104
12.4 Logical operators . . . 104
12.4.1 NOT operator . . . 105
12.4.2 AND operator . . . 105
12.4.3 OR operator . . . 105
12.4.4 XOR operator . . . 105
12.5 String operators . . . 106
12.5.1 String indexing . . . 106
12.5.2 String concatenation operator . . . 107
12.6 Aggregate operators . . . 107
12.6.1 Aggregate indexing . . . 107
12.6.2 Intersection operator . . . 108
12.6.3 Union operator . . . 109

©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

12.6.4 Difference operator . . . 109
12.6.5 Subset operator . . . 110
12.6.6 Superset operator . . . 111
12.6.7 Query expression . . . 111
12.7 References . . . 112
12.7.1 Simple references . . . 113
12.7.2 Prefixed references . . . 113
12.7.3 Attribute references . . . 114
12.7.4 Group references . . . 115
12.8 Function call . . . 116
12.9 Aggregate initializer . . . 117
12.10 Complex entity instance construction operator . . . 118
12.11 Type compatibility . . . 119
12.12 Select data types in expressions . . . 120
12.12.1 Select data types in unary expressions . . . 120
12.12.2 Select data types in binary expressions . . . 120
12.12.3 Select data types in ternary expressions . . . 121

13 Executable statements . . . 121
13.1 Null (statement) . . . 121
13.2 Alias statement . . . 122
13.3 Assignment . . . 122
13.3.1 Assignment statement . . . 122
13.3.2 Assignment compatibility . . . 123
13.4 Case statement . . . 126
13.5 Compound statement . . . 127
13.6 Escape statement . . . 127
13.7 If . . . Then . . . Else statement . . . 127
13.8 Procedure call statement . . . 128
13.9 Repeat statement . . . 128
13.9.1 Increment control . . . 129
13.9.2 While control . . . 130
13.9.3 Until control . . . 130
13.10 Return statement . . . 131
13.11 Skip statement . . . 131

14 Built-in constants . . . 132
14.1 Constant e . . . 132
14.2 Indeterminate . . . 132
14.3 False . . . 132
14.4 Pi . . . 132
14.5 Self . . . 132
14.6 True . . . 133
14.7 Unknown . . . 133

15 Built-in functions . . . 133
15.1 Abs - arithmetic function . . . 133
15.2 ACos - arithmetic function . . . 133
15.3 ASin - arithmetic function . . . 133
15.4 ATan - arithmetic function . . . 134
15.5 BLength - binary function . . . 134
15.6 Cos - arithmetic function . . . 134
15.7 Exists - general function . . . 135

 ©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

15.8 Exp - arithmetic function 135
15.9 Format - general function 135
15.9.1 Symbolic representation 136
15.9.2 Picture representation 137
15.9.3 Standard representation 138
15.10 HiBound - arithmetic function 138
15.11 HiIndex - arithmetic function 138
15.12 Length - string function 139
15.13 LoBound - arithmetic function 139
15.14 Log - arithmetic function 140
15.15 Log2 - arithmetic function 140
15.16 Log10 - arithmetic function 140
15.17 LoIndex - arithmetic function 140
15.18 NVL - null value function 141
15.19 Odd - arithmetic function 141
15.20 RolesOf - general function 142
15.21 Sin - arithmetic function 143
15.22 SizeOf - aggregate function 143
15.23 Sqrt - arithmetic function 143
15.24 Tan - arithmetic function 144
15.25 TypeOf - general function 144
15.26 UsedIn - general function 147
15.27 Value - arithmetic function 148
15.28 Value_in - membership function 148
15.29 Value_unique - uniqueness function 149

16 Built-in procedures 149
16.1 Insert 149
16.2 Remove 150

Annex A (normative) EXPRESS language syntax 151
A.1 Tokens 151
A.1.1 Keywords 151
A.1.2 Character classes 153
A.1.3 Lexical elements 154
A.1.4 Remarks 154
A.1.5 Interpreted identifiers 154
A.2 Grammar rules 155
A.3 Cross reference listing 159

Annex B (normative) Determination of the allowed entity instantiations 166
B.1 Formal approach 166
B.2 Supertype and subtype constraint operators 167
B.2.1 OneOf 168
B.2.2 And 168
B.2.3 AndOr 168
B.2.4 Precedence of operators 168
B.3 Interpreting the possible complex entity data types 168

Annex C (normative) Instance limits imposed by the interface specification 181

Annex D (normative) EXPRESS-G: A graphical subset of EXPRESS 185
D.1 Introduction and overview 185

©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

D.2 Definition symbols 185
D.2.1 Symbol for simple data types 186
D.2.2 Symbols for constructed data types 187
D.2.3 Symbols for defined data types 188
D.2.4 Symbols for entity data types 188
D.2.5 Symbols for subtype_constraints 188
D.2.6 Symbols for functions and procedures 188
D.2.7 Symbols for rules 189
D.2.8 Symbols for schemas 189
D.3 Relationship symbols 189
D.4 Composition symbols 190
D.4.1 Page references 190
D.4.2 Inter-schema references 191
D.5 Entity level diagrams 191
D.5.1 Role names 191
D.5.2 Cardinalities 192
D.5.3 Constraints 192
D.5.4 Constructed and defined data types 193
D.5.5 Entity data types 193
D.5.6 Inter-schema references 197
D.6 Schema level diagrams 198
D.7 Complete EXPRESS-G diagrams 198
D.7.1 Complete entity level diagram 199
D.7.2 Complete schema level diagram 200

Annex E (normative) Protocol implementation conformance statement (PICS) 201
E.1 EXPRESS language parser 201
E.2 EXPRESS-G editing tool 201

Annex F (normative) Information object registration 203
F.1 Document identification 203
F.2 Syntax identification 203

Annex G (normative) Generating a single schema from multiple schemas 204
G.1 Introduction 204
G.2 Fundamental concepts 204
G.3 Name munging 206
G.3.1 Name clashes 206
G.3.2 Identifiers as strings 206
G.4 Stage 1: multi-schema to intermediate schema conversion 207
G.4.1 Introduction 207
G.4.2 Primary population 207
G.4.3 Secondary population 209
G.4.4 Prune 216
G.4.5 Schema names and versions 222
G.5 Stage 2: convert intermediate schema to ISO 10303-11:1994 223
G.5.1 Introduction 223
G.5.2 Initialisation 223
G.5.3 Conversion of extensible constructed data types 223
G.5.4 Conversion of subtype constraints 228
G.5.5 Conversion of abstract entity and generalized types 231
G.5.6 Conversion of attributes renamed in a redeclaration 233

 ©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Annex H (informative) Relationships 235
H.1 Relationships via attributes 235
H.1.1 Simple relationship 236
H.1.2 Collective relationship 238
H.1.3 Distributive relationship 239
H.1.4 Inverse attribute 240
H.2 Subtype/supertype relationships 241

Annex J (informative) EXPRESS models for EXPRESS-G illustrative examples 242
J.1 Example single schema model 242
J.2 Relationship sampler 243
J.3 Simple subtype/supertype tree 244
J.4 Attribute redeclaration 244
J.5 Multi-schema models 245

Annex K (informative) Deprecated features of EXPRESS 248

Annex L (informative) Examples of the new EXPRESS constructs 249
L.1 Product management example 249

Bibliography 251

Index 252

Figures

Figure B.1 — EXPRESS-G diagram of schema for example 1 on page 171 172
Figure B.2 — EXPRESS-G diagram of schema for example 2 on page 174 174
Figure B.3 — EXPRESS-G diagram of schema for example 3 on page 176 177
Figure D.1 — Complete entity level diagram of the example in J.1 on page 242 (Page 1 of 2) 186
Figure D.2 — Complete entity level diagram of the example in J.1 on page 242 (Page 2 of 2) 186
Figure D.3 — Symbols for EXPRESS simple data types 186
Figure D.4 — Symbol for EXPRESS generic_entity data type 187
Figure D.5 — Symbols for EXPRESS constructed data types 187
Figure D.6 — Abbreviated symbols for the EXPRESS constructed data types when used as the representation of defined data types 187
Figure D.7 — Example of alternative methods for representing an enumeration data type 187
Figure D.8 — Symbols for EXPRESS extensible constructed data types 188
Figure D.9 — Symbol for EXPRESS defined data type 188
Figure D.10 — Symbol for an EXPRESS entity data type 188
Figure D.11 — Symbol for an EXPRESS subtype_constraint 188
Figure D.12 — Symbol for a schema 189
Figure D.13 — Relationship line styles 189
Figure D.14 — Partial entity level diagram illustrating relationship directions from the example in J.2 on page 243 (Page 1 of 1) 190
Figure D.15 — Composition symbols: page references 190
Figure D.16 — Composition symbols: inter-schema references 191
Figure D.17 — Complete entity level diagram of the example in J.2 on page 243 (Page 1 of 1) 192
Figure D.18 — Extensible select data type diagram 193
Figure D.19 — Symbol for denoting an ABSTRACT SUPERTYPE if the abstract constraint is defined within a SUBTYPE_CONSTRAINT 195
Figure D.20 — Symbol denoting an ABSTRACT ENTITY 195

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Figure D.21 — Example of the TOTAL_OVER coverage constraint 196
Figure D.22 — Complete entity level diagram of the inheritance graph from the example in J.3 on page 244 (Page 1 of 1) 196
Figure D.23 — Complete entity level diagram of the example in J.4 on page 245 showing attribute redeclarations in subtypes (Page 1 of 1) 197
Figure D.24 — Complete entity level diagram of the `top` schema of example 1 on page 245 illustrating inter-schema references (Page 1 of 1) 197
Figure D.25 — Complete schema level diagram of example 1 on page 245 (Page 1 of 1) . 198
Figure D.26 — Complete schema level diagram of example 2 on page 246 (Page 1 of 1) . 199

Tables

Table 1 — EXPRESS keywords 14
Table 2 — EXPRESS reserved words which are operators 14
Table 3 — EXPRESS reserved words which are constants 15
Table 4 — EXPRESS reserved words which are function names 15
Table 5 — EXPRESS reserved words which are procedure names 15
Table 6 — EXPRESS symbols 16
Table 7 — The use of data types 36
Table 8 — Supertype expression operator precedence 59
Table 9 — Scope and identifier defining items 79
Table 10 — Operator precedence 93
Table 11 — Pattern matching characters 103
Table 12 — NOT operator 105
Table 13 — AND operator 105
Table 14 — OR operator 105
Table 15 — XOR operator 106
Table 16 — Intersection operator – operand and result types 108
Table 17 — Union operator – operand and result types 110
Table 18 — Difference operator – operand and result types 110
Table 19 — Subset and superset operators - operand types 111
Table 20 — Example symbolic formatting effects 137
Table 21 — Picture formatting characters 137
Table 22 — Example picture formatting effects 137

©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Foreword

ISO (the International Organization for Standardization) 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.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 2.

The main task of technical committees is to prepare International Standards. Draft International Standards adopted by the 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.

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO shall not be held responsible for identifying any or all such patent rights.

ISO 10303–11 was prepared by Technical Committee ISO/TC 184, *Industrial automation systems and integration*, Subcommittee SC 4, *Industrial data*.

This second edition of ISO 10303–11 constitutes a minor revision of the first edition (ISO 10303-11:1994), which is provisionally retained in order to support continued use and maintenance of implementations based on the first edition and to satisfy the normative references of other parts of ISO 10303. This second edition also incorporates the Technical Corrigendum ISO 10303–11:1994/Cor.1:1999(E).

ISO 10303 is organized as a series of parts, each published separately. The structure of ISO 10303 is described in ISO 10303–1.

Each part of ISO 10303 is a member of one of the following series: description methods, implementation methods, conformance testing methodology and framework, integrated generic resources, integrated application resources, application protocols, abstract test suites, application interpreted constructs, and application modules. This part is a member of the description methods series.

A complete list of parts of ISO 10303 is available from the Internet:

`<http://www.tc184-sc4.org/titles/>`

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

0 Introduction

0.1 General

ISO 10303 is an International Standard for the computer-interpretable representation of product information and for the exchange of product data. The objective is to provide a neutral mechanism capable of describing products throughout their life cycle. This mechanism is suitable not only for neutral file exchange, but also as a basis for implementing and sharing product databases, and as a basis for archiving.

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.

The changes that lead to this edition were driven by requirements from multi-schema specifications. The new concepts constitute an architecture for extensible data models. The following keywords have been added to this edition:

— BASED_ON;

— END_SUBTYPE_CONSTRAINT;

— EXTENSIBLE;

— GENERIC_ENTITY;

— RENAMED;

— SUBTYPE_CONSTRAINT;

— TOTAL_OVER;

— WITH.

Schemas that contain these words as EXPRESS identifiers become invalid under this edition. Else, the modifications that are incorporated in this edition are upwardly compatible with the previous edition.

0.2 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 ISO 10303. 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;

 ©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

— 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.

NOTE 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.

NOTE 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 ISO 10303. 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.

©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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 specifies a language by which aspects of product data can be defined. The language is called EXPRESS.

This part of ISO 10303 also specifies 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 that allow an unambiguous data definition and specification of constraints on the data defined.

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

— 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.

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

2 Normative references

The following referenced documents are indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

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

ISO/IEC 8824-1:2002, *Information technology — Abstract Syntax Notation One (ASN.1): Specification of basic notation*

ISO/IEC 10646:2003, *Information technology — Universal Multiple-Octet Coded Character Set (UCS)*

3 Terms and definitions

3.1 Terms defined in ISO 10303–1

For the purposes of this part of ISO 10303, the following terms defined in ISO 10303–1 apply.

— Conformance requirement;

— Context;

— Data;

— Data specification language;

— Information;

— Information model;

— PICS proforma;

3.2 Terms defined in ISO/IEC 10646

For the purposes of this part of ISO 10303, the following term defined in ISO/IEC 10646 applies.

— Graphic character.

NOTE This definition includes only those characters in ISO/IEC 10646 that have a defined visual representation; this explicitly excludes any cells that are empty or crosshatched.

3.3 Other terms and definitions

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

 ©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

3.3.1
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.

3.3.2
complex entity (data type) instance
a named complex entity data type value. The name of a complex entity instance is used for referencing the instance.

3.3.3
complex entity (data type) value
a unit of data that represents a unit of information within the class defined by a complex entity data type. It is a member of the domain established by this complex entity data type.

3.3.4
constant
a named unit of data from a specified domain. The value cannot be modified.

3.3.5
data type
a domain of values.

3.3.6
entity
a class of information defined by common properties.

3.3.7
entity data type
a representation of an entity. An entity data type establishes a domain of values defined by common attributes and constraints.

3.3.8
entity (data type) instance
a named entity data type value. The name of an entity instance is used for referencing the instance.

3.3.9
(single) entity (data type) value
a unit of data which represents a unit of information within the class defined by an entity data type. It is a member of the domain established by this entity data type.

3.3.10
instance
a named value.

3.3.11
multi-leaf complex entity (data type)
a complex entity data type that consists of more than one entity data types that do not have further subtypes within this complex entity data type.

©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

3.3.12
multi-leaf complex entity (data type) instance
a named multi-leaf complex entity data type value. The name of a multi-leaf complex entity instance is used for referencing the instance.

3.3.13
multi-leaf complex entity (data type) value
a unit of data that represents a unit of information within the class defined by a multi-leaf complex entity data type. It is a member of the domain established by this multi-leaf complex entity data type.

3.3.14
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.

3.3.15
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.

3.3.16
population
a collection of entity data type instances.

3.3.17
primary schema
a schema in a group of interrelated schemas that form a, possibly cyclic, directed graph. A primary schema is a schema of interest. There can be one or more primary schemas within the graph, where the other schemas in the graph are only there to support the primary schema. The primary schema has a special role in the conversion of a shortform schema into a longform schema (see G).

3.3.18
root schema
a schema in a group of interrelated schemas that form a, possibly cyclic, directed graph. The root schema is not the target of any interface specification, but all the other schemas can be reached from the root schema. The root schema can be considered to be representative of the graph. The root schema has a special role in the conversion of a shortform schema into a longform schema (see G).

3.3.19
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.

3.3.20
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.

 ©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

3.3.21
token
a non-decomposable lexical element of a language.

3.3.22
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 clauses 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 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.

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

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.

©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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. Transient members may violate uniqueness constraints, global rules, and where rules. This standard does not define the behaviour of functions or procedures when instances that violate these constraints are passed to them as actual parameters.

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.

©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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 L.1 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/IEC 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;

 ©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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

EXAMPLE 1 The syntax for a string type is as follows:

Syntax:

```
311 string_type = STRING [ width_spec ] .
341 width_spec = '(' width ')' [ FIXED ] .
340 width = numeric_expression .
```

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

EXAMPLE 2 Following the syntax given in example 1, 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/IEC 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;

— `\x9`, `\xA`, and `\xD` represent the characters in positions 9, 10, and 13 respectively of 00, plane 00, group 00 of ISO/IEC 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.

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

EXAMPLE 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 use only the characters in the following character set: characters allocated to cells 09, 0A, 0D, the graphic characters lying in the range 20 to 7E of ISO/IEC 10646, and the special character `\n` signifying the newline. This set of characters is called the EXPRESS character set. A member of this set is referred to by the cell of the standard in which this character is allocated; these cell numbers are specified in hexadecimal. The printable characters from this set (cells 21–7E of ISO/IEC 10646) 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:

The character set thus specified is an abstract character set; it is independent of its representation in an implementation.

NOTE 1 ISO/IEC 6429 specifies semantics for the characters in positions 09, 0A and 0D in ISO/IEC 10646. This part of ISO 10303 does not require the semantics prescribed in ISO/IEC 6429; neither does it preclude them.

NOTE 2 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:

```
124 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 7).

Syntax:

```
128 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' .
```


©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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:

```
137 special = not_paren_star_quote_special | '(' | ')' | '*' | '''' .
132 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 shall not 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. The notation `\s` may 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

The characters in cells 09, 0A and 0D shall be treated as whitespace, unless within a string literal. The notation `\xn` where `n` is one of the characters 9, A, and D shall be used to represent these 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. Both forms of remark may be associated with an identified construct using a remark tag.

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

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:

```
145 embedded_remark = ’(*’ [ remark_tag ] { ( not_paren_star { not_paren_star } ) |
                      lparen_then_not_lparen_star | ( ’*’ { ’*’ } ) |
                      not_rparen_star_then_rparen | embedded_remark } ’*)’ .
147 remark_tag = ’"’ remark_ref { ’.’ remark_ref } ’"’ .
148 remark_ref = attribute_ref | constant_ref | entity_ref | enumeration_ref |
                 function_ref | parameter_ref | procedure_ref | rule_label_ref |
                 rule_ref | schema_ref | subtype_constraint_ref | type_label_ref |
                 type_ref | variable_ref .
131 not_paren_star = letter | digit | not_paren_star_special .
128 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’ .
124 digit = ’0’ | ’1’ | ’2’ | ’3’ | ’4’ | ’5’ | ’6’ | ’7’ | ’8’ | ’9’ .
133 not_paren_star_special = not_paren_star_quote_special | ’’’’ .
132 not_paren_star_quote_special = ’!’ | ’"’ | ’#’ | ’$’ | ’%’ | ’&’ | ’+’ | ’,’ |
                                   ’-’ | ’.’ | ’/’ | ’:’ | ’;’ | ’<’ | ’=’ | ’>’ |
                                   ’?’ | ’@’ | ’[’ | ’\’ | ’]’ | ’^’ | ’_’ | ’‘’ |
                                   ’{’ | ’|’ | ’}’ | ’~’ .
129 lparen_then_not_lparen_star = ’(’ { ’(’ } not_lparen_star { not_lparen_star } .
130 not_lparen_star = not_paren_star | ’)’ .
138 not_rparen_star_then_rparen = not_rparen_star { not_rparen_star } ’)’ { ’)’ } .
135 not_rparen_star = 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 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:

```
149 tail_remark = ’--’ [ remark_tag ] { \a | \s | \x9 | \xA | \xD } \n .
147 remark_tag = ’"’ remark_ref { ’.’ remark_ref } ’"’ .
148 remark_ref = attribute_ref | constant_ref | entity_ref | enumeration_ref |
                 function_ref | parameter_ref | procedure_ref | rule_label_ref |
                 rule_ref | schema_ref | subtype_constraint_ref | type_label_ref |
                 type_ref | variable_ref .
```

 ©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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

7.1.6.3 Remark tag

A remark may be related to a named item, that is, an item which is associated with an identifier, by placing a remark tag as the first sequence of characters within the remark. The remark tag shall immediately follow the pair of characters that is used to identify the remark. The remark tag itself consists of a reference to the identifier declared in the item that is enclosed by quotation mark signs.

Syntax:

```
147 remark_tag = '"' remark_ref { '.' remark_ref } '"' .
148 remark_ref = attribute_ref | constant_ref | entity_ref | enumeration_ref |
                 function_ref | parameter_ref | procedure_ref | rule_label_ref |
                 rule_ref | schema_ref | subtype_constraint_ref | type_label_ref |
                 type_ref | variable_ref .
```

Rules and restrictions:

a) The `remark_ref` shall follow the visibility rules defined in 10.2.

b) A qualified remark reference shall use the visibility rules defined in 10.2 in the following manner: the reference to the left of the '.' shall identify the scope in which to find the reference to the right.

NOTE A qualified remark reference is a remark reference that uses the '.'-notation (see syntax rule 147 above).

c) If a remark reference is not found according to the visibility rules identified above, the remark is not to be associated with any item.

d) A tagged remark that contains other tagged remarks (via nesting) shall be associated completely (including nested remarks) with the item referenced. The inner tagged remarks shall be associated with their identified items also.

e) If both the nested remark and the enclosing remark refer to the same identified item, the nested remark shall be associated with that item twice; once inside the enclosing remark, and once directly.

EXAMPLE 1 The tagged remark in the example below refers to the attribute `attr` in the scope of `ent`.

```
ENTITY ent;
  attr: INTEGER;
END_ENTITY;
(*"ent.attr" The attr attribute ... *)
```

EXAMPLE 2 The reference to a SCHEMA `my_second_schema` may in a tagged remark be followed by any identifier that is declared directly within the scope of that SCHEMA, for example, by the name of the FUNCTION `a_complicated_function` in the example below.

```
SCHEMA my_second_schema;
...
FUNCTION a_complicated_function;
```

©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
...
END_FUNCTION;
(*"my_second_schema.a_complicated_function" This complicated function ... *)
...
END_SCHEMA;
```

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.

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

Table 1 – EXPRESS keywords

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

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			

©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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.

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.

©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Table 6 – EXPRESS symbols

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

Syntax:

```
143 simple_id = letter { letter | digit | ’_’ } .
128 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’ .
124 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:

```
251 literal = binary_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:

```
139 binary_literal = ’%’ bit { bit } .
123 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 A valid binary literal

```
%0101001100
```


©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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:

```
141 integer_literal = digits .
125 digits = digit { digit } .
124 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 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:

```
142 real_literal = integer_literal |
                   ( digits '.' [ digits ] [ 'e' [ sign ] digits ] ) .
125 digits = digit { digit } .
124 digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' .
304 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.

EXAMPLE 1 Valid real literals

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

EXAMPLE 2 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
```


Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

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 apostrophes ('). An apostrophe within a simple string literal is represented by two consecutive apostrophes. An encoded string literal is a four octet encoded representation of each character in a sequence of ISO/IEC 10646 characters enclosed in quotation marks ("). The encoding is defined as follows:

a) first octet = ISO/IEC 10646 group in which the character is defined;

b) second octet = ISO/IEC 10646 plane in which the character is defined;

c) third octet = ISO/IEC 10646 row in which the character is defined;

d) fourth octet = ISO/IEC 10646 cell in which the character is defined.

The sequence of octets shall identify one of the valid characters of ISO/IEC 10646.

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

Syntax:

```
310 string_literal = simple_string_literal | encoded_string_literal .
144 simple_string_literal = \q { ( \q \q ) | not_quote | \s | \x9 | \xA | \xD } \q .
134 not_quote = not_paren_star_quote_special | letter | digit | '(' | ')' | '*' .
132 not_paren_star_quote_special = '!' | '"' | '#' | '$' | '%' | '&' | '+' | ',' |
                                   '-' | '.' | '/' | ':' | ';' | '<' | '=' | '>' |
                                   '?' | '@' | '[' | '\' | ']' | '^' | '_' | '`' |
                                   '{' | '|' | '}' | '~' .
128 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' .
124 digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' .
140 encoded_string_literal = '"' encoded_character { encoded_character } '"' .
126 encoded_character = octet octet octet octet .
136 octet = hex_digit hex_digit .
127 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.

EXAMPLE 1 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

 ©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

EXAMPLE 2 Invalid simple string literals

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

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

EXAMPLE 3 Valid encoded string literals

`"00000041"`
Reads A

`"000000C5"`
Reads Å

`"0000795E00006238"`
These are the Japanese ideographs 神戸 for Kobe.

EXAMPLE 4 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:

```
255 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.

©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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:

```
261 number_type = NUMBER .
```

EXAMPLE Since we may not know the context of `size`, we do not know how to correctly represent it. The size of the crowd at a football game, for example, 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:

```
278 real_type = REAL [ '(' precision_spec ')' ] .
268 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.

©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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:

```
241 integer_type = INTEGER .
```

EXAMPLE 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:

```
256 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:

```
182 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 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 that are permitted as part of a string value are those characters allocated to cells 09, 0A, 0D and the

©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

graphic characters lying in the ranges 20 to 7E and A0 to 10FFFE of ISO/IEC 10646.

Syntax:

```
311 string_type = STRING [ width_spec ] .
341 width_spec = '(' width ')' [ FIXED ] .
340 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.

EXAMPLE 1 The following defines a varying length string; values of which have no defined maximum length.

```
string1 : STRING;
```

EXAMPLE 2 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);
```

EXAMPLE 3 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:

```
181 binary_type = BINARY [ width_spec ] .
341 width_spec = '(' width ')' [ FIXED ] .
340 width = numeric_expression .
```


©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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 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. The AGGREGATE data type is the generalization of these four aggregation data types (see 9.5.3.1).

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

  EXAMPLE 1 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 2 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 3 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.

©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS

Not for Resale

EXAMPLE 4 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 5 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:

```
175 array_type = ARRAY bound_spec OF [ OPTIONAL ] [ UNIQUE ] instantiable_type .
185 bound_spec = '[' bound_1 ':' bound_2 ']' .
183 bound_1 = numeric_expression .
184 bound_2 = numeric_expression .
```

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 (that is, 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

 ©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

comparisons between indeterminate (?) values result in UNKNOWN, so the uniqueness constraint is not violated.

EXAMPLE 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:

```
250 list_type = LIST [ bound_spec ] OF [ UNIQUE ] instantiable_type .
185 bound_spec = '[' bound_1 ':' bound_2 ']' .
183 bound_1 = numeric_expression .
184 bound_2 = numeric_expression .
```

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 (that is, not instance equal to) every other element in the same list value.

EXAMPLE 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;
```


Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

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:

```
180 bag_type = BAG [ bound_spec ] OF instantiable_type .
185 bound_spec = '[' bound_1 ':' bound_2 ']' .
183 bound_1 = numeric_expression .
184 bound_2 = numeric_expression .
```

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 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.

 ©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Syntax:

```
303 set_type = SET [ bound_spec ] OF instantiable_type .
185 bound_spec = '[' bound_1 ':' bound_2 ']' .
183 bound_1 = numeric_expression .
184 bound_2 = numeric_expression .
```

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 (that is, not instance equal to) every other element in the same set value.

EXAMPLE 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 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.

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

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

FUNCTION my_unique(c: AGGREGATE OF GENERIC): LOGICAL;
  (*"my_unique" 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 subclause 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:

```
152 entity_ref = entity_id .
```


©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Rules and restrictions:

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

EXAMPLE 1 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:

```
162 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 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. The real number means volume, rather than some other thing that 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 may only be used to provide underlying representations of defined data types (see 9.1).

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

8.4.1 Enumeration data type

An ENUMERATION data type has as its domain a set of names. The extent of this set of names is determined depending on the type of ENUMERATION data type. The following types of ENUMERATION data types are distinguished:

— ENUMERATION that is extensible;

— ENUMERATION that is extending an extensible ENUMERATION, in other words: that is based on an extensible ENUMERATION;

— ENUMERATION that is neither extensible nor extending.

The names that an ENUMERATION declares are the only valid values of the ENUMERATION data type. Each name in the domain is referred to as an enumeration item, and is designated by an `enumeration_id`.

An ENUMERATION data type that is neither an extensible nor an extending ENUMERATION data type has as its domain the ordered set of enumeration items in its declaration.

An extensible ENUMERATION data type has as its domain the set of the enumeration items in its declaration plus the union of the sets of enumeration items comprising the domains of all extending enumeration data types. An extensible enumeration data type is a generalization of the enumeration data types that are based on it. An extensible ENUMERATION data type is specified using the EXTENSIBLE reserved word.

An extending enumeration data type has as its domain the set of enumeration items in its declaration plus the enumeration items specified directly, not via extension, in the extensible enumeration data type upon which it is based. An extending ENUMERATION data type is specified using the BASED_ON reserved word.

An enumeration data type may be both an extensible and an extending enumeration data type. An extensible ENUMERATION may be specified without enumeration items and may be based on another extensible ENUMERATION and not specify any enumeration items extending that base ENUMERATION. The based-on relationship is transitive, that is, an extending enumeration is still based on the highest level extensible enumeration even through several levels of based-on relationships; it includes all enumeration items of both the intermediate extensible enumerations and the highest level extensible enumeration.

NOTE 1 An extensible enumeration that is extended two or more times in a single context may have a larger domain than its extensions, in which case it really is a generalization.

NOTE 2 In the first edition of this part of ISO 10303 the ordering of the enumeration items defined a value ordering. In this edition of this part of ISO 10303, no ordering is defined except as noted in rule (d) below. This is to allow for extensible enumeration data types, in which the ordering of the extensions cannot be determined.

Syntax:

```
213 enumeration_type = [ EXTENSIBLE ] ENUMERATION [ ( OF
                       enumeration_items ) | enumeration_extension ] .
211 enumeration_items = '(' enumeration_id { ',' enumeration_id } ')' .
209 enumeration_extension = BASED_ON type_ref [ WITH enumeration_items ] .
```

 ©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Rules and restrictions:

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

b) An enumeration data type may be extended only if the reserved word EXTENSIBLE is specified in its definition.

c) The `type_ref` in an `enumeration_extension` shall refer to an extensible enumeration type.

d) For comparison purposes, the ordering of the values of an enumeration data type that is neither extensible nor extending may 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, and so on.

e) There is no ordering of the values of an extensible enumeration or an extending enumeration.

f) An enumeration that is neither extensible nor extending shall specify enumeration items as its domain.

g) An enumeration that is not extensible, but extending shall specify enumeration items that extend the domain of the extensible enumeration upon which it is based.

h) Two different ENUMERATION data types may contain the same `enumeration_id` in their sets of names. If these enumeration data types are not extensions to the same extensible enumeration data type, their `enumeration_id`s describe different concepts, even though their local names may be identical. In this case, any reference to the `enumeration_id` (for example, in an expression) shall be qualified with the data type identifier to ensure that the reference is unambiguous. The reference then appears as: `type_id.enumeration_id`.

NOTE 3 A `type_id` for use in qualifying an `enumeration_id` is always defined as an ENUMERATION data type.

EXAMPLE 1 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 data 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 is qualified by the data type name, for example, `car_can_move.left`.

i) An extensible enumeration and its extensions define a domain consisting of `enumeration_id`s. Within that domain all occurrences of the same `enumeration_id` designate the same value even when the `enumeration_id` is specified in multiple enumeration data types contributing to that domain.

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

EXAMPLE 2 The following EXPRESS results in a single enumeration item named `red` as both `stop_light` and `canadian_flag` extend the domain of colour.

```
TYPE colour = EXTENSIBLE ENUMERATION; END_TYPE;

TYPE stop_light = ENUMERATION BASED_ON colour WITH (red, yellow,
green); END_TYPE;

TYPE canadian_flag = ENUMERATION BASED_ON colour WITH (red,
white); END_TYPE;
```

j) The type declaration which declares the enumeration data type shall not include a domain (where) rule.

NOTE 4 The above rules ensure that a defined data type names an enumeration data type, and the defined data type is not a specialization of the enumeration data type.

EXAMPLE 3 The following example shows how an extensible enumeration may be used to model a context dependent concept of approval. `general_approval`, as its name suggests, is the most general concept of approval explicitly defining only two values. By declaring `general_approval` to be an extensible enumeration allows it to take on context dependent values in schemas which declare extensions to it. If used to represent the domain of an attribute, the allowed values of the attribute are context dependent.

```
SCHEMA s1;

TYPE general_approval = EXTENSIBLE ENUMERATION OF (approved, rejected);
END_TYPE;

END_SCHEMA;

SCHEMA s2;

USE FROM s1 (general_approval);

TYPE domain2_approval = EXTENSIBLE ENUMERATION BASED_ON general_approval WITH (pending);
END_TYPE;

END_SCHEMA;

SCHEMA s3;

USE FROM s1 (general_approval);

TYPE domain3_approval = EXTENSIBLE ENUMERATION BASED_ON general_approval WITH (cancelled);
END_TYPE;

END_SCHEMA;

SCHEMA s4;

USE FROM s2 (domain2_approval);

REFERENCE FROM s3 (domain3_approval);

TYPE specific_approval = ENUMERATION BASED_ON domain2_approval WITH (rework);
END_TYPE;
```

 ©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
END_SCHEMA;

SCHEMA s5;

USE FROM s1 (general_approval);

TYPE redundant_approval = ENUMERATION BASED_ON general_approval WITH (approved);
END_TYPE;

END_SCHEMA;
```

In the context of schema s1:

— general_approval has the domain (approved, rejected).

In the context of schema s2:

— general_approval has the domain (approved, rejected, pending);

— domain2_approval has the domain (approved, rejected, pending).

In the context of schema s3:

— general_approval has the domain (approved, rejected, cancelled);

— domain3_approval has the domain (approved, rejected, cancelled).

In the context of schema s4:

— general_approval has the domain (approved, rejected, pending, cancelled, rework);

— domain2_approval has the domain (approved, rejected, pending, rework);

— domain3_approval has the domain (approved, rejected, cancelled);

— specific_approval has the domain (approved, rejected, pending, rework).

In the context of schema s5:

— general_approval has the domain (approved, rejected).

— redundant_approval has the domain (approved, rejected).

8.4.2 Select data type

A SELECT data type defines a data type that enables a choice among several named data types. The SELECT data type is a generalization of these named data types in its domain. The defined type for which the SELECT data type is the underlying representation, may add constraints on its domain by declaring local rules. A SELECT data type may be EXTENSIBLE or not.

A SELECT data type that is neither EXTENSIBLE nor extending has as its domain the union of the domains of the named data types in its select list.

An EXTENSIBLE SELECT data type has as its domain the union of the domains of the named data types in its own select list, plus the union of the domains of all extending SELECT data types. An EXTENSIBLE SELECT data type is specified using the EXTENSIBLE reserved word.

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

An extending SELECT data type has as its domain the named data types in its own select list, plus the named data types specified directly, not via extension, in the EXTENSIBLE SELECT data type upon which it is based. An extending SELECT data type is specified using the BASED_ON reserved word.

A SELECT data type may be both an EXTENSIBLE and an extending SELECT data type. An EXTENSIBLE SELECT may be specified without a select list and may be based on another EXTENSIBLE SELECT and not specify a select list extending that base SELECT.

An EXTENSIBLE SELECT data type and only an EXTENSIBLE one may be constrained to have only ENTITY instances in its domain by using the reserved word GENERIC_ENTITY. In this case, all SELECT elements shall be generic-entity elements, where generic-entity element is defined as being either an ENTITY data type or a SELECT of generic-entity elements. All extensions of that SELECT data type shall be generic-entity SELECT data types and shall specify the GENERIC_ENTITY reserved word.

Syntax:

```
302 select_type = [ EXTENSIBLE [ GENERIC_ENTITY ] ] SELECT
                  [ select_list | select_extension ] .
301 select_list = '(' named_types { ',' named_types } ')' .
300 select_extension = BASED_ON type_ref [ WITH select_list ] .
```

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.

c) A SELECT data type may be extended only if the reserved word EXTENSIBLE is specified in its definition.

d) The `type_ref` in an `select_extension` shall refer to an extensible select type.

e) A SELECT that is neither extensible nor extending shall specify a non-empty select list as its domain.

f) A SELECT that is not extensible, but extending shall specify a non-empty select list that extends the domain of the extensible SELECT upon which it is based.

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.

EXAMPLE 1 If *a* and *b* are subtypes of *c*, and if they are related by an ANDOR expression, and if there is a data type defined by `SELECT (a,b)`, then it may be that the value of the SELECT data type is an *a* and a *b* at the same time.

EXAMPLE 2 A choice must be made among several types of things in a given context.

```
TYPE attachment_method = EXTENSIBLE SELECT(nail, screw);
END_TYPE;

TYPE permanent_attachment = SELECT BASED ON attachment_method WITH (glue, weld);
END_TYPE;
```

 ©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
ENTITY nail;
  length    : REAL;
  head_area : REAL;
END_ENTITY;

ENTITY screw;
  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 an attachment method. The initial attachment method describes temporary attachment methods. These methods are then extended to add permanent attachment methods. A value of a `wall_mounting` will have a `using` attribute that is a value of one of `nail`, `screw`, `glue`, or `weld`.

8.5 Generalized data types

Syntax:

```
223 generalized_types = aggregate_type | general_aggregation_types |
                        generic_entity_type | 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 aggregation data type is a generalization of the aggregation data types that relaxes some of the constraints normally applied to aggregation data types. A GENERIC_ENTITY data type is a generalization of all ENTITY data types and a subtype of the GENERIC data type. The GENERIC_ENTITY data type is used to constrain EXTENSIBLE SELECT data types (see 8.4.2). It is also a valid parameter data type for the formal parameters in functions and procedures and for the type representation of attributes in ABSTRACT ENTITY data type declarations. (see 9.5.3.3). All of these data types are defined in 9.5.3.

8.6 Data type usage classification

In this clause (8), EXPRESS data types are classified according to their nature as: simple data types, aggregation data types, constructed data types, named data types, and generalized data types. This subclause defines the classification of data types according to their usage.

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Data types are used in six different ways in EXPRESS:

— as the data types of the elements of aggregation data types;

— as the members of a select list in defining or extending a select data type;

— as the underlying types of data types;

— as the data types of of attributes of entity data types;

— as the data types of constants;

— as the data types of formal parameters and local variables in functions and procedures.

In addition, there are several special usages of entity data types, specified in 9, that apply to no other class of data type, and are not considered here.

Data types are classified according to their usage as follows:

— instantiable data types are used as representations for aggregation elements and as the data types of constants;

— parameter data types are used as representations for explicit and derived attributes and for formal parameters, function results, and local variables in functions and procedures;

— underlying data types are used as representations for defined data types;

— named data types are used as the members of a select list — the possible representations of a value of the SELECT data type.

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	d
Simple Data Types	•	•	•	
Aggregation Data Types	•	•	•	
Named Data Types	•	•	•[1)]	•
Constructed Data Types			•	
Generalized Data Types		•		

a) Instantiable data types — representation of aggregation elements and constants.

b) Parameter data types — representation of an explicit or derived attribute, a formal parameter, local variable, or function result.

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

d) Named data types — possible representations of a SELECT data type.

[1)] Only the defined data type from the named data types may be used as an underlying data type.

Named data types are exactly as specified in 8.3. Instantiable data types, parameter data types, and underlying data types are defined below.

 ©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

8.6.1 Instantiable data types

Instantiable data types are used as the representations of constants, as the representations of elements of aggregation data types, and as the representations of attributes of non-abstract entity data types (see 9.2.1).

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

Syntax:

```
240 instantiable_type = concrete_types | entity_ref .
193 concrete_types = aggregation_types | simple_types | type_ref .
```

Rules and restrictions:

a) The data type of a constant shall not be an abstract entity data type (see 9.4).

b) The data type of every attribute of a non-abstract entity data type shall be, or shall be redclared to be, an instantiable data type (see 9.2.1).

8.6.2 Parameter data types

Parameter data types are used as the representation of attributes of entity data types or 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 instantiable data types and the generalized data types. That is, all EXPRESS data types are parameter data types (but constructed data types can only be used as the defined data types that are based on them).

Syntax:

```
266 parameter_type = generalized_types | named_types | simple_types .
223 generalized_types = aggregate_type | general_aggregation_types |
                        generic_entity_type | generic_type .
```

Rules and restrictions:

a) Any parameter type that satisfies the specifications for an instantiable type is considered to be an instantiable type for those usages in which an instantiable type is required.

b) Any general aggregation type (see 9.5.3.5) whose base type is an instantiable data type is considered to be an instantiable type.

NOTE A syntactic construct such as `ARRAY[1:3] OF REAL` satisfies two syntactic productions — `aggregation_type` and `general_aggregation_type`. It is considered to be instantiable no matter which production it is required to satisfy in the syntax.

8.6.3 Underlying data types

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

©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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

Syntax:

```
332 underlying_type = concrete_types | constructed_types .
193 concrete_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 principal capabilities of EXPRESS are found in the following declarations:

— Type;

— Entity;

— Subtype_constraint;

— 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 subclauses, 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. An exception to this is for constructed data types where the defined data type is used to name the constructed data type and is, in fact, not a specialization of the constructed data type even in the case of a SELECT data type constrained by a where rule.

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`.

 ©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Syntax:

```
327 type_decl = TYPE type_id '=' underlying_type ';' [ where_clause ] END_TYPE ';' .
332 underlying_type = concrete_types | constructed_types .
```

Rules and restrictions:

TYPE declarations shall not result in cyclic type definitions.

EXAMPLE 1 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:

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

Each `domain_rule` may be given a rule label.

NOTE 2 When given, rule labels may be used in remark tags (see 7.1.6.3) or to identify rules to implementations, for example, in documentation, error reports, and enforcement specifications. The labelling of rules for these purposes 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 2 A defined data type could be created which constrains the underlying integer data type to allow only positive integers.

©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
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).

An entity instance may only be created in EXPRESS by means of an entity constructor (see 9.2.6) or a complex entity construction operator (see 12.10).

Syntax:

```
206 entity_decl = entity_head entity_body END_ENTITY ';' .
207 entity_head = ENTITY entity_id subsuper ';' .
204 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.

 ©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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 the essential traits, qualities, or properties of an entity. 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 H 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:

```
215 explicit_attr = attribute_decl { ',' attribute_decl } ':' [ OPTIONAL ]
                    parameter_type ';' .
177 attribute_decl = attribute_id | redeclared_attribute .
266 parameter_type = generalized_types | named_types | simple_types .
```

NOTE 1 The syntax for `redeclared_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.

©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

NOTE 2 If the data type of an explicit attribute is an extensible enumeration type for which no enumeration items are specified, the entity can not be instantiated unless some extension of the enumeration type is declared with at least one enumeration item. If the data type of an explicit attribute is an extensible select type for which no select element is specified, the entity can not be instantiated unless some extension of the select type is declared with at least one named type.

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 (?).

c) An explicit attribute shall neither directly nor indirectly be declared to be of the data type GENERIC.

NOTE 3 This is in spite of the fact that the syntax production allows such a declaration.

NOTE 4 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. The case where an attribute is meaningless for some instances is properly modelled by subtyping (see 9.2.3).

NOTE 5 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 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:

```
200 derived_attr = attribute_decl ':' parameter_type ':=' expression ';' .
177 attribute_decl = attribute_id | redeclared_attribute .
266 parameter_type = generalized_types | named_types | simple_types .
```

NOTE 1 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.

 ©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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.

c) A derived attribute shall neither directly nor indirectly be declared to be of the data type GENERIC.

NOTE 2 This is in spite of the fact that the syntax production allows such a declaration.

EXAMPLE 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 H 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. In case of ambiguities due to identical attribute names in the subtype/supertype graph of the referencing entity, the name of the explicit attribute shall be preceded by the name of the entity that originally declares the attribute.

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

©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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.

NOTE 2 If the inverted attribute is represented by a non-unique aggregation data type, that is, a list or array that 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.

NOTE 3 If the inverted attribute is represented by a unique aggregation data type, that is, a list or array that 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, that is, a list or array that 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:

```
248 inverse_attr = attribute_decl ':' [ ( SET | BAG ) [ bound_spec ] OF ] entity_ref
                   FOR [ entity_ref '.' ] attribute_ref ';' .
177 attribute_decl = attribute_id | redeclared_attribute .
185 bound_spec = '[' bound_1 ':' bound_2 ']' .
183 bound_1 = numeric_expression .
184 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 one of the following:

— the current entity being declared;

— a supertype of the entity being declared;

— a defined data type based on a select data type containing one of the above;

— an aggregation type whose fundamental type is one of the above.

 ©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

c) The entity referred to in the inverse attribute declaration may be a subtype of the entity that declared the direct relationship, in which case the inverse attribute only contains instances of that subtype. Subtypes referred to in this way shall not redeclare the direct attribute as a derived attribute.

d) If the name of the explicit attribute in the entity defining the direct relationship is not unique within the subtype/supertype graph of that entity, the name of the entity data type shall be used to qualify the name of this explicit attribute following the FOR keyword.

e) An inverse attribute shall neither directly nor indirectly be declared to be of the data type GENERIC.

NOTE 5 This is in spite of the fact that the syntax production allows such a declaration.

EXAMPLE 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 (it may, for example, be 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.

NOTE When given, rule labels may be used in remark tags (see 7.1.6.3) or to identify rules to implementations, for example, in documentation, error reports, and enforcement specifications. The labelling of rules for these purposes is encouraged.

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

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:

```
333 unique_clause = UNIQUE unique_rule ';' { unique_rule ';' } .
334 unique_rule = [ rule_label_id ':' ] referenced_attribute { ','
                  referenced_attribute } .
280 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.

EXAMPLE 1 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.

EXAMPLE 2 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;
```


©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
  ...
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:

```
338 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 1 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.

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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 2 Consider this variation of example 1.

```
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 (that is, subtype/supertype) relationship between the entities in which the subtype inherits the properties (that is, 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:

```
312 subsuper = [ supertype_constraint ] [ subtype_declaration ] .
```

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 edges represent the subtype/supertype relationships. Following the directed `SUBTYPE OF` edges leads to supertypes while following the directed `SUBTYPE OF` edges in the reverse leads from supertypes to subtypes.

 ©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Rules and restrictions:

a) The supertype constraint, if present, shall precede the subtype constraint, 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, that means, the subtype/supertype graph shall be 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). An instance of an entity data type that is a subtype is an instance of each of its supertypes.

Syntax:

```
318 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. An entity is implicitly specified to be a supertype if it is named in a SUBTYPE declaration of at least one other entity.

Syntax:

```
319 supertype_constraint = abstract_entity_declaration |
                           abstract_supertype_declaration | supertype_rule .
164 abstract_entity_declaration = ABSTRACT .
166 abstract_supertype_declaration = ABSTRACT SUPERTYPE [ subtype_constraint ] .
313 subtype_constraint = OF '(' supertype_expression ')' .
320 supertype_expression = supertype_factor { ANDOR supertype_factor } .
321 supertype_factor = supertype_term { AND supertype_term } .
323 supertype_term = entity_ref | one_of | '(' supertype_expression ')' .
263 one_of = ONEOF '(' supertype_expression { ',' supertype_expression } ')' .
322 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.

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

EXAMPLE 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:

a) 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).

b) 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 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; that is, there is only one value for this attribute in an instance of this ENTITY data type.

 ©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

9.2.3.4 Attribute redeclaration

An attribute declared in a supertype may 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 the following general ways:

— the attribute may be given a different name;

— the data type of the attribute may be changed to a specialization of the original data type (see 9.2.7);

EXAMPLE 1 A NUMBER data type attribute may be changed to an INTEGER data type or to a REAL data type.

— if the original data type of the attribute is a defined data type based on a select, the type may be changed to either another select, which identifies a subset or specialization of items from the original select list, or a specialization of one of the items from the original select list;

— 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;

— an attribute in the supertype may be given a new identifier in the subtype. The new identifier obeys all the scope and visibility rules defined in clause 10 for an identifier for an attribute of the subtype whose declaration includes this redeclaration; but the identifier always refers to the original attribute in the supertype.

NOTE 1 The declaration of a new identifier does not remove the old identifier from the name scope. The old identifier remains available in this ENTITY data type and in any subtypes that are declared for this ENTITY data type.

Syntax:

```
279 redeclared_attribute = qualified_attribute [ RENAMED attribute_id ] .
275 qualified_attribute = SELF group_qualifier attribute_qualifier .
232 group_qualifier = '\' entity_ref .
179 attribute_qualifier = '.' attribute_ref .
```

Rules and restrictions:

a) The data type in the redeclaration shall be the same as, or a specialization of, the data type of the attribute declared in the supertype. The rules for specialization in 9.2.7 apply.

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

c) If the data type used to define the original attribute was constrained by a where rule, the data type used to define the redeclared attribute should be constrained such that the domain of the redeclared attribute is a subset of the domain of the original attribute.

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

d) The `group_qualifier` in the `qualified_attribute` syntax shall identify either the entity data type in which the attribute was originally declared, or an entity data type that redeclares the attribute from another supertype.

e) 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.

f) If the attribute is given a new identifier, that identifier shall not be the same as the identifier for any attribute in any supertype of the current ENTITY data type.

NOTE 2 A WHERE rule that is specified against the original attribute remains applicable to redeclarations of that attribute (see 9.2.3.5).

EXAMPLE 2 Some geometry systems use floating point coordinates while others work in an integer coordinate space. The concepts of GENERIC_ENTITY and RENAMED enable specifications of general applicability and specialization to domain specific uses.

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

ENTITY integer_point
  SUBTYPE OF (point);
  SELF\point.x RENAMED integer_x : INTEGER;
  SELF\point.y RENAMED integer_y : INTEGER;
END_ENTITY;

ENTITY line ABSTRACT;
  start : GENERIC_ENTITY;
  end   : GENERIC_ENTITY;
END_ENTITY;

ENTITY integer_point_line;
  SUBTYPE OF (line);
  SELF\line.start RENAMED integer_start : integer_point;
  SELF\line.end   RENAMED integer_end   : integer_point;
END_ENTITY;
```

EXAMPLE 3 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;
```

 ©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

EXAMPLE 4 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;
  (*"distance" Compute the shortest distance between two points *)
END_FUNCTION;

FUNCTION normal(p1, p2, p3 : point) : vector;
  (*"normal" 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 all instances of a supertype applies to all instances of its subtypes. 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.

NOTE 1 If the constraints specified in two (or more) entity data types conflict, there can be no valid

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

instance that contains those entity data types.

EXAMPLE 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 2 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 Abstract entity data type

EXPRESS allows for the declaration of ENTITY data types that are not intended to be directly instantiated and may only be instantiated through their subtypes. An ABSTRACT ENTITY data type may declare explicit and derived attributes whose data types are generalized data types (see 8.5). These may then be redeclared as being of an instantiable data type in subtypes of the ABSTRACT ENTITY data type. If a subtype of an ABSTRACT ENTITY data type is itself an ABSTRACT ENTITY data type, it need not redeclare the uninstantiable inherited attributes to be of instantiable data types. In a subtype of an ABSTRACT ENTITY data type, that is not

 ©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

itself an ABSTRACT ENTITY data type, no inherited or directly declared attributes shall be of an uninstantiable data type.

Type labels (see 9.5.3.4) may be used to ensure that two or more attributes whose data types are generalized data types are the same data types at time of invocation.

Rules and restrictions:

a) An ABSTRACT ENTITY data type declaration includes the ABSTRACT keyword in the ENTITY data type declaration, but not the SUPERTYPE keyword (see 9.2.5.1 for ABSTRACT SUPERTYPE).

b) An ABSTRACT ENTITY data type is not instantiable unless it is part of a complex ENTITY data type where all attributes whose data types are generalized data types have been redeclared to be of an instantiable data type.

NOTE 1 Rule (b) implies that every ABSTRACT ENTITY data type satisfies the ABSTRACT SUPERTYPE constraint (see 9.2.5.1).

NOTE 2 The redeclaration can occur directly in the instantiable subtype or in one of its supertypes (see 9.2.4).

EXAMPLE In a general approval model, we may wish to identify that a collection of things may be approved. This model could then be used in a number of other schemas and refined to identify the actual items approved.

```
ENTITY general_approval ABSTRACT;
  approved_items : BAG OF GENERIC_ENTITY;
  status         : approval_status;
END_ENTITY;
```

9.2.5 Subtype/supertype constraints

An instance of an entity data type that 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 H.2).

Syntax:

```
319 supertype_constraint = abstract_entity_declaration |
                           abstract_supertype_declaration | supertype_rule .
164 abstract_entity_declaration = ABSTRACT .
166 abstract_supertype_declaration = ABSTRACT SUPERTYPE [ subtype_constraint ] .
313 subtype_constraint = OF '(' supertype_expression ')' .
320 supertype_expression = supertype_factor { ANDOR supertype_factor } .
321 supertype_factor = supertype_term { AND supertype_term } .
323 supertype_term = entity_ref | one_of | '(' supertype_expression ')' .
263 one_of = ONEOF '(' supertype_expression { ',' supertype_expression } ')' .
322 supertype_rule = SUPERTYPE subtype_constraint .
```

It is possible to specify constraints on which subtype/supertype graphs may be instantiated. These constraints may be specified in the declaration of the supertype by means of the SUPERTYPE clause. They may also be specified as separate rules, by means of SUBTYPE_CONSTRAINT declarations (see 9.7).

©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

NOTE 1 In order that existing schemas remain valid, the declaration of subtype/supertype constraints that use the keywords ONEOF, ANDOR, or AND within the declaration of an entity, as described in this sub-clause, remains valid under this edition 2 of EXPRESS. However, its use is deprecated, and its removal is planned in future editions. The use of the SUBTYPE_CONSTRAINT (see 9.7) is encouraged instead.

The meaning of a SUBTYPE_CONSTRAINT is the set of constraints stated in the `supertype_expression`. A SUBTYPE_CONSTRAINT can contain any number of AND constraints and ONEOF constraints. Each is interpreted as a separate constraint.

In addition, when it appears in the statement of some more complex constraint, each ONEOF, AND, and ANDOR expression is interpreted as a set of instances of the supertype. In interpreting the `supertype_expression`, the following rules apply:

— an entity data type name appearing anywhere in a `supertype_expression` is interpreted as the set of entity instances constituting the entire population of that data type, just as in a global rule (see 9.6);

— the result of an expression in a `supertype_expression` is interpreted as a set of instances of the supertype, as specified for ONEOF, AND, and ANDOR below.

Although the final result of the `supertype_expression` in the SUBTYPE_CONSTRAINT is, therefore, a set of entity instances, that set has no significance. That is, the result of the whole `supertype_expression` does not state any constraint because it does not necessarily include all instances of the supertype, and may include instances to which none of the stated constraints applies.

NOTE 2 Therefore, independent constraints can be connected by ANDOR which only adds instances to the (meaningless) overall result of the `supertype_expression`.

Annex B provides the formal approach to determining the potential combinations of subtype/supertype that may be instantiated under the several possible constraints that are described below. That is, Annex B characterizes the supertype populations that satisfy the whole SUBTYPE_CONSTRAINT.

9.2.5.1 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 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 (for example, land-based or water-based). The entity data type for a vehicle must not be independently instantiated.

```
ENTITY vehicle
  ABSTRACT SUPERTYPE;
END_ENTITY;

ENTITY land_based
```

 ©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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

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

9.2.5.2 ONEOF

The ONEOF constraint states that the populations of the operands in the ONEOF list are mutually exclusive; no instance of any of the populations of the operands in the ONEOF list shall appear in the population of any other operand in the ONEOF list.

Syntax:

```
263 one_of = ONEOF '(' supertype_expression { ',' supertype_expression } ')' .
320 supertype_expression = supertype_factor { ANDOR supertype_factor } .
321 supertype_factor = supertype_term { AND supertype_term } .
323 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. When an ONEOF constraint occurs as an operand in another constraint, it represents the set of entity instances that is the union of the populations of the operands in the ONEOF list.

NOTE The phrase `ONEOF(a,b,c)` reads in natural language as “an instance shall consist of at most one of the entity data types `a,b,c`.”

EXAMPLE 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;
```

©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

9.2.5.3 ANDOR

An instance of the population designated by an ANDOR expression may be an instance of either operand population or both. Thus, ANDOR does not state a constraint. Within a complex constraint, the ANDOR represents the set of entity instances that is the union of the populations of the operand expressions.

NOTE ANDOR is only used in constructing a union of populations for a more complex constraint. The phrase `b ANDOR c` means "all instances of type `b` and all instances of type `c`, including any that happen to be instances of both".

EXAMPLE 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;
```

9.2.5.4 AND

AND states the constraint that the populations specified by the two operands shall be identical. That is, any instance of the left operand population shall also be an instance of the right operand population, and any instance of the right operand population shall also be an instance of the left operand population.

When the AND expression occurs as an operand in a complex constraint, it represents either of its operand populations; they are identical.

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

EXAMPLE 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
```


©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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

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

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

9.2.5.5 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.

Table 8 – Supertype expression operator precedence

Precedence	Operators
1	( ) ONEOF
2	AND
3	ANDOR

EXAMPLE 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;
```

9.2.5.6 Default constraint between subtypes

If no supertype constraint is mentioned in the declaration of an entity, the subtypes (if any) shall be mutually inclusive, that is, 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 The model in example 9.2.5.3 is equivalent to using the default construction, as:

```
ENTITY person
  ...
END_ENTITY;

ENTITY employee
  SUBTYPE OF (person);
```

©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
  ...
END_ENTITY;

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

9.2.6 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 by a shallow copy. A shallow copy is one in which an entity instance is copied by reference, that is, the attribute is a reference to the instance used as an actual parameter, a simple type is value copied and aggregates have their elements shallow copied. The constructor shall only provide the attributes which are explicit in a particular entity declaration.

Syntax:

```
205 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 (that is, the parentheses shall always be 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

©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

values for these redeclared attributes. These values are ignored in favour of the derived value.

EXAMPLE 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;
```

9.2.7 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;

— an ENTITY is a specialization of a GENERIC_ENTITY;

— an EXTENSIBLE GENERIC_ENTITY SELECT is a specialization of a GENERIC_ENTITY;

— a SELECT data type that contains only ENTITY data types is a specialization of a GENERIC_ENTITY;

— the aggregation data types are specializations of AGGREGATE;

— a SELECT of a, b, c is a specialization of a SELECT of d, e, f if a, b, c are specializations of d, e, f;

— a SELECT of a, b, c is a specialization of a supertype if a, b, c are subtypes of the supertype;

— 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`;

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

- 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 constructed data type that is based on an extensible constructed data type is a specialization of that extensible constructed data type;

- 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. A SCHEMA may undergo changes within a development or standardization environment. To support the capability of identifying a particular version of a SCHEMA a schema version identifier is defined. This part of ISO 10303 does not prescribe the format of the schema version identifier except to define it to be a string literal. No other construct defined in this part of ISO 10303 makes reference to this schema version identifier, and no method for managing schema changes using schema version identifiers is specified in this part of ISO 10303. If two schemas with the same name have different schema version identifiers, they will nevertheless be treated as the same schema.

NOTE For schemas defined in ISO 10303, the use of an information object identifier is specified that includes a version identifier. The meaning of the object identifer is defined in ISO/IEC 8824-1, and is described in ISO 10303-1. The use of this object identifier as a schema version identifier is encouraged.

EXAMPLE 1 Geometry may be the name of a schema that contains declarations of points, curves, surfaces, and other related data types.

EXAMPLE 2 There may be multiple versions of schemas, and the language version identifier may be included as well. The support_resource_schema in this example uses the information object identifier as defined in ISO/IEC 8842-1 and described in ISO 10303-1.

```
SCHEMA geometry_schema 'version_1';
END_SCHEMA;

SCHEMA geometry_schema 'version_2';
END_SCHEMA;

SCHEMA support_resource_schema '{ISO standard 10303 part(41) object(1) version(8)}';
END_SCHEMA;

SCHEMA support_resource_schema '{ISO standard 10303 part(41) object(1) version(9)}';
END_SCHEMA;
```


©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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:

```
296 schema_decl = SCHEMA schema_id [ schema_version_id ] ';' schema_body
                  END_SCHEMA ';' .
298 schema_version_id = string_literal .
295 schema_body = { interface_specification } [ constant_decl ]
                  { declaration | rule_decl } .
242 interface_specification = reference_clause | use_clause .
199 declaration = entity_decl | function_decl | procedure_decl |
                  subtype_constraint_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 initialization of another named dependent constant. The declaration of constants shall be acyclic.

NOTE The requirement for acyclic constant declarations is to ensure that there is always a valid initialization, which does not have to be in the order of declaration.

Syntax:

```
195 constant_decl = CONSTANT constant_body { constant_body } END_CONSTANT ';' .
194 constant_body = constant_id ':' instantiable_type ':=' expression ';' .
240 instantiable_type = concrete_types | entity_ref .
```

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 assignment compatible with the specified base type.

EXAMPLE 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;
```

©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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, and so on, 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:

```
220 function_decl = function_head algorithm_head stmt { stmt } END_FUNCTION ';' .
221 function_head = FUNCTION function_id [ '(' formal_parameter
                    { ';' formal_parameter } ')' ] ':' parameter_type ';' .
218 formal_parameter = parameter_id { ',' parameter_id } ':' parameter_type .
266 parameter_type = generalized_types | named_types | simple_types .
173 algorithm_head = { declaration } [ constant_decl ] [ local_decl ] .
199 declaration = entity_decl | function_decl | procedure_decl |
                  subtype_constraint_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.

NOTE Local variables that are declared to be entity data types may be assigned instances from the formal parameters. Changes to these local variables will affect the formal parameter, since the assignment is by reference. Changes to formal parameters are not reflected to the point of invocation,

 ©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

according to the above rule, therefore great care should be taken when attempting to return these local variables.

e) Functions may modify local variables or parameters that are declared in an outer scope, that is, 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:

```
271 procedure_decl = procedure_head algorithm_head { stmt } END_PROCEDURE ';' .
272 procedure_head = PROCEDURE procedure_id [ '(' [ VAR ] formal_parameter
                     { ';' [ VAR ] formal_parameter } ')' ] ';' .
218 formal_parameter = parameter_id { ',' parameter_id } ':' parameter_type .
266 parameter_type = generalized_types | named_types | simple_types .
173 algorithm_head = { declaration } [ constant_decl ] [ local_decl ] .
199 declaration = entity_decl | function_decl | procedure_decl |
                  subtype_constraint_decl | type_decl .
```

Rules and restrictions:

Procedures may modify local variables or parameters that are declared in an outer scope, that is, 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:

```
218 formal_parameter = parameter_id { ',' parameter_id } ':' parameter_type .
266 parameter_type = generalized_types | named_types | simple_types .
```

EXAMPLE 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 generalized data types (AGGREGATE, the general aggregation data types, GENERIC, and GENERIC_ENTITY) (see 8.5) are used to allow generalization of the data types used to represent

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

the formal parameters of functions and procedures. AGGREGATE, general aggregation data types (see 9.5.3.5), and generic entity data types (GENERIC_ENTITY) may in addition be used to declare explicit or derived attributes of ABSTRACT ENTITY data types. 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 that is defined as 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.

NOTE Type labels (see 9.5.3.4) may be used to ensure that, at the time of invocation, two or more parameters that are represented by AGGREGATE data types 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.

When an explicit or derived attribute in the data type declaration of an ABSTRACT ENTITY is represented by an AGGREGATE data type, this attribute shall in the subtypes of the entity be declared as an ARRAY, BAG, LIST, or SET.

Syntax:

```
171 aggregate_type = AGGREGATE [ ':' type_label ] OF parameter_type .
329 type_label = type_label_id | type_label_ref .
266 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 the result type of a function, or as the type of a local variable within a function or procedure, or as the representation of an explicit or derived attribute in an ABSTRACT ENTITY data type declaration.

b) If an AGGREGATE data type is used as the result type of a function, or as the type of a local variable within a function or procedure, type label references are required for this usage. The type label refernces shall refer to type labels declared by the formal parameters (see 9.5.3.4).

c) If an AGGREGATE data type is used as the representation of an explicit or derived attribute in an ABSTRACT ENTITY data type declaration, the attribute shall be redeclared to be an ARRAY, BAG, LIST, or SET in the non-abstract subtypes of this entity data type.

EXAMPLE 1 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 REAL;
               scalar: REAL): AGGREGATE:intype OF REAL;
  LOCAL
    result : AGGREGATE:intype OF REAL := input;
  END_LOCAL;

  IF SIZEOF(['BAG','SET'] * TYPEOF(input)) > 0 THEN
```

 ©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
    REPEAT i := LOINDEX(input) TO HIINDEX(input);
      result := result - input[i];          -- remove the original
      result := result + scalar*input[i];   -- insert the scaled
    END_REPEAT;
  ELSE
    REPEAT i := LOINDEX(input) TO HIINDEX(input);
      result[i] := scalar*input[i];
    END_REPEAT;
  END_IF;

  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.

NOTE Type labels (see 9.5.3.4) may be used to ensure that, at the time of invocation, two or more parameters that are represented by GENERIC data types 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.

Syntax:

```
231 generic_type = GENERIC [ ':' type_label ] .
329 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 the result type of a function, or as the type of a local variable within a function or procedure.

b) If a GENERIC data type is used as the result type of a function, or as the type of a local variable within a function or procedure, type label references are required for this usage. The type label references shall refer to type labels declared by the formal parameters (see 9.5.3.4).

EXAMPLE 1 This example shows a general purpose 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,
```


Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

```
                  a.j + b.j,
                  a.k + b.k);
      RETURN(vr);
    END_IF;
  END_IF;
  RETURN (?); --"add" if we receive input that is invalid, return a no-value
END_FUNCTION;
```

9.5.3.3 Generic entity data type

A GENERIC_ENTITY data type is a generalization of all entity data types.

When a procedure or function with a formal parameter that is defined to be a GENERIC_ENTITY data type is invoked, the actual parameter passed shall be an entity instance. The operations that can be performed depend on the data type of the actual parameter.

NOTE Type labels (see 9.5.3.4) may be used to ensure that, at the time of invocation, two or more parameters that are represented by GENERIC_ENTITY data types 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.

When an explicit or derived attribute in the data type declaration of an ABSTRACT ENTITY is represented by a GENERIC_ENTITY data type, this attribute shall in the subtypes of the entity be declared to be a specific entity data type.

Syntax:

```
230 generic_entity_type = GENERIC_ENTITY [ ':' type_label ] .
329 type_label = type_label_id | type_label_ref .
```

Rules and restrictions:

a) A GENERIC_ENTITY data type shall only be used as a formal parameter type of a function or procedure, or as the result type of a function, or as the type of a local variable within a function or procedure, or as the representation of an explicit or derived attribute in an ABSTRACT ENTITY data type declaration.

b) If a GENERIC_ENTITY data type is used as the result type of a function, or as the type of a local variable within a function or procedure, type label references are required for this usage. The type label references shall refer to type labels declared by the formal parameters (see 9.5.3.4).

EXAMPLE 1 The following function checks whether a particular instance `sample` is referenced from two instances `type1` and `type2` of known entity data types. Declaring the formal parameter `sample` to be of type GENERIC_ENTITY allows instances of any entity data types to be valid input to this function.

```
FUNCTION check_relating (type1  : instance_of_type_1;
                         type2  : instance_of_type_2;
                         sample : GENERIC_ENTITY): BOOLEAN;
  RETURN ((type1 IN USEDIN(sample, ''))
          AND
          (type2 IN USEDIN(sample, '')));
END_FUNCTION;
```


©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

9.5.3.4 Type labels

Type labels shall be used to relate the data type of an attribute or actual parameter at invocation to the data types of other attributes or actual parameters, local variables, or the return type of a function. Type labels are declared for:

— AGGREGATE, GENERIC_ENTITY, and GENERIC data types within the formal parameter declaration of a function or procedure. They may then be referenced by AGGREGATE, GENERIC_ENTITY, or GENERIC data types in the formal parameter declaration or the local variable declaration of a FUNCTION or PROCEDURE, or the declaration of the returned data type of a FUNCTION.

— AGGREGATE and GENERIC_ENTITY data types within the declaration of explicit or derived attributes of abstract entities. They may then be referenced by AGGREGATE or GENERIC_ENTITY data types in the remainder of the entity data type declaration.

Syntax:

```
329 type_label = type_label_id | type_label_ref .
```

Rules and restrictions:

a) The first occurrence of a type label in the declaration of a formal parameter or an explicit or derived attribute declares that type label; subsequent occurrences of that type label are references to the first occurrence.

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 that declares the type label.

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

d) The data types of attributes that refer via a type label to an attribute data type shall be identical to the attribute data type that declares the type label.

EXAMPLE This example indicates how type labels may be used for type compatibility checks at invocation of a function.

```
ENTITY a;
...
END_ENTITY;

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

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

...
```


Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

```
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 12.10
  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 are in clause 15.

9.5.3.5 General aggregation data types

General aggregation data types form part of the class of types called generalized data types. A general aggregation data type represents a generalization of the corresponding aggregation data types (ARRAY, BAG, LIST and SET) that allows the element type to be a generalized data type. That is, a `general_list_type` is a generalization of the `list_type`, as specified below, and similarly for array, bag and set types.

Syntax:

```
224 general_aggregation_types = general_array_type | general_bag_type |
                                general_list_type | general_set_type .
225 general_array_type = ARRAY [ bound_spec ] OF [ OPTIONAL ] [ UNIQUE ]
                         parameter_type .
185 bound_spec = '[' bound_1 ':' bound_2 ']' .
183 bound_1 = numeric_expression .
184 bound_2 = numeric_expression .
266 parameter_type = generalized_types | named_types | simple_types .
226 general_bag_type = BAG [ bound_spec ] OF parameter_type .
227 general_list_type = LIST [ bound_spec ] OF [ UNIQUE ] parameter_type .
229 general_set_type = SET [ bound_spec ] OF parameter_type .
```

The general aggregation data type generalizes the corresponding aggregation data type in the following ways:

— a general array type can be specified without specifying 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.

— the underlying data type may be a GENERIC, GENERIC ENTITY, AGGREGATE, or general aggregation data type when used as formal parameter to a function or procedure, or GENERIC ENTITY, AGGREGATE, or general aggregation data type when used in an ABSTRACT ENTITY. This is specified more formally below.

 ©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Let G be the general aggregation data type, and EG be the (generalized) data type of its elements. For a formal parameter to a function or procedure, let A be the data type of the corresponding actual parameter; for an attribute of an abstract entity data type, let A be the data type of the redeclared attribute of a non-abstract subtype: and in either case, let EA be the data type of the elements of A.

— If G is a general array type, A shall be an ARRAY data type. If the index range of G is specified, the index range of A shall be identical.

— If G is a general bag type, A shall be a BAG data type. If the bounds of G are specified, the bounds of A shall be identical.

— If G is a general list type, A shall be a LIST data type. If the bounds of G are specified, the bounds of A shall be identical.

— If G is a general set type, A shall be a SET data type. If the bounds of G are specified, the bounds of A shall be identical.

— If EG is any generalized data type, then EA shall conform to EG as specified for that generalized data type in this subclause (9.5).

— If EG is not a generalized data type, then EA shall be assignment compatible with EG, as specified in 13.3.

EXAMPLE 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:

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

Initialization of local variables

A local variable may appear in the initialization of another dependent local variable. The declaration of dependent local variables shall be acyclic. If no initializer is given the indeterminate (?) value is assigned to the local variable.

NOTE 1 The requirement for acyclic local variable declarations is to ensure that there is always a valid initialization, which does have to be in the order of declaration.

NOTE 2 Since indeterminate (?) is compatible with all data types explicit initialization with the indeterminate (?) value is allowed.

©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

EXAMPLE 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 (these are 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:

```
291 rule_decl = rule_head algorithm_head { stmt } where_clause END_RULE ';' .
292 rule_head = RULE rule_id FOR '(' entity_ref { ',' entity_ref } ')' ';' .
173 algorithm_head = { declaration } [ constant_decl ] [ local_decl ] .
199 declaration = entity_decl | function_decl | procedure_decl |
                  subtype_constraint_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.

d) For a collection of instances to be in a valid domain, all global rules specified for that domain must be asserted. This includes asserting rules for entity data types for which there are no instances in the collection of instances under test.

   NOTE A global rule may be written to ensure that at least one instance of a specified data type exists. Not checking this rule due to no instances of the specified entity data type would not enforce the required semantics.

 ©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

EXAMPLE 1 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 2 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 that contains the set of all instances of the named entity data type in the domain. This set of entity instances is governed by the rule.

Syntax:

```
267 population = entity_ref .
```

Rules and restrictions:

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

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

EXAMPLE 3 Assuming the following declaration:

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

the implicitly declared variable would be:

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

9.7 Subtype constraints

The concepts of subtypes and supertypes are specified in 9.2.3. The concept of subtype/supertype constraints is specified in 9.2.5. It is possible to specify constraints on which subtype/supertype graphs may be instantiated outside the declaration of an entity. These may be specified by declaring a SUBTYPE_CONSTRAINT.

Syntax:

```
315 subtype_constraint_decl = subtype_constraint_head subtype_constraint_body
                              END_SUBTYPE_CONSTRAINT ';' .
316 subtype_constraint_head = SUBTYPE_CONSTRAINT subtype_constraint_id FOR
                              entity_ref ';' .
314 subtype_constraint_body = [ abstract_supertype ] [ total_over ]
                              [ supertype_expression ';' ] .
165 abstract_supertype = ABSTRACT SUPERTYPE ';' .
326 total_over = TOTAL_OVER '(' entity_ref { ',' entity_ref } ')' ';' .
320 supertype_expression = supertype_factor { ANDOR supertype_factor } .
321 supertype_factor = supertype_term { AND supertype_term } .
323 supertype_term = entity_ref | one_of | '(' supertype_expression ')' .
263 one_of = ONEOF '(' supertype_expression { ',' supertype_expression } ')' .
```

The SUBTYPE_CONSTRAINT is used to specify the following constraints on the possible supertype/subtype instantiation:

— that the supertype is abstract and should only be instantiated through its subtypes;

— that a collection of the subtypes for the supertype provides a total coverage; that is, if a total coverage is specified, an instance of any subtype of the supertype shall also be an instance of at least one of the subtypes specified in the TOTAL_OVER specification;

— the relationship between some of the subtypes.

Each of these different areas will be discussed in more detail in the following subclauses. 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.

 ©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

9.7.1 Abstract supertype constraint

The ABSTRACT SUPERTYPE declaration specified in 9.2.5.1 may also be declared in a SUBTYPE_CONSTRAINT.

Rules and restrictions:

An ABSTRACT SUPERTYPE is defined by a SUBTYPE_CONSTRAINT for the supertype including the ABSTRACT SUPERTYPE keywords.

EXAMPLE In a general classification model, we may wish to identify an entity called `class`, which in this context is instantiable. In a more specific model, we may wish to use the `class` ENTITY but constrain it such that it may only be instantiated through its locally declared subtypes.

```
SCHEMA general_classification_model;

ENTITY class;
  name : class_name;
END_ENTITY;

END_SCHEMA;

SCHEMA specific_classification_model;

USE FROM general_classification_model;

ENTITY class_of_facility
  SUBTYPE OF (class);
END_ENTIY;

ENTITY class_of_organization
  SUBTYPE OF (class);
END_ENTITY;

SUBTYPE_CONSTRAINT independent_classification FOR class;
  ABSTRACT SUPERTYPE;
  ONEOF(class_of_facility, class_of_organization);
END_SUBTYPE_CONSTRAINT;

END_SCHEMA;
```

9.7.2 Total coverage subtypes

A total coverage TOTAL_OVER constraint specifies that every instance of the supertype shall be an instance of one or more of a given set of subtypes. That is, for a given context, the domain of the supertype is exactly equal to the set union of the domains of the named subtypes.

EXAMPLE 1 The concept person is totally covered by the concepts of male and female. There may be other concepts, but each person is either male or female. Therefore, we can say that person is totally covered by male and female.

If two or more subtype constraints specify TOTAL_OVER constraints for the same entity data type, these TOTAL_OVER constraints are considered in combination, this means, `TOTAL_OVER (a,b)` and `TOTAL_OVER (c,d)` shall both be satisfied.

Rules and restrictions:

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

a) All subtypes specified in the one or several TOTAL_OVER constraints for a given supertype shall be direct subtypes of that supertype.

b) Instances of other subtypes, however those subtypes are defined or constrained, shall also be instances of one or more of the subtypes named in the TOTAL_OVER specification.

c) Since a supertype may have more than one context, it may also have more than one TOTAL_OVER constraints.

EXAMPLE 2 The following specifies that each `person` is either a `male` or a `female`. The example says nothing about the relationship between `male` and `female`, and it is possible to create an instance that is both `male` and `female`. The subtype `employee` shall always be combined with the concepts of `male` and `female` and can not be instantiated independently.

```
ENTITY person;
  name : personal_name;
END_ENTITY;

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

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

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

SUBTYPE_CONSTRAINT person_sex FOR person;
  ABSTRACT SUPERTYPE;
  TOTAL_OVER (male, female);
END_SUBTYPE_CONSTRAINT;
```

9.7.3 Overlapping subtypes and their specification

Two or more direct subtypes of a particular supertype may be allowed to have overlapping instantiations for a particular context. The SUBTYPE_CONSTRAINT specification may be used to specify how a particular group of direct subtypes are related.

9.7.3.1 ONEOF

The ONEOF constraint specified in 9.2.5.2 may be declared in a SUBTYPE_CONSTRAINT.

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

```
ENTITY pet
  name : pet_name;
  ...
END_ENTITY;
```

 ©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

```
SUBTYPE_CONSTRAINT separate_species FOR pet;
  ABSTRACT SUPERTYPE;
  ONEOF(cat, rabbit, dog, ... );
END_SUBTYPE_CONSTRAINT;

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

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

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

9.7.3.2 ANDOR

The ANDOR constraint specified in 9.2.5.3 may be declared in a SUBTYPE_CONSTRAINT.

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

```
ENTITY person
  ...
END_ENTITY;

SUBTYPE_CONSTRAINT employee_may_be_student FOR person;
  employee ANDOR student;
END_SUBTYPE_CONSTRAINT;

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

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

9.7.3.3 AND

The AND constraint specified in 9.2.5.4 may be declared in a SUBTYPE_CONSTRAINT.

EXAMPLE 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;
  ...
END_ENTITY;

SUBTYPE_CONSTRAINT no_mixing FOR person;
  SUPERTYPE OF
```

©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
    (ONEOF(male, female) AND
     ONEOF(citizen, alien));
END_SUBTYPE_CONSTRAINT;

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;
```

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 the 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.

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

 ©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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	•	•
subtype constraint	•	•
type	•	•
type label		•
variable		•

NOTE 1 The identifier is an implicitly declared variable within the defined scope of the declaration.
NOTE 2 The variable is only implicitly declared when an increment control is specified.
NOTE 3 An implicit variable declaration is made for all entities which are constrained by the rule.

schema via two, or more, routes which share the same initial declaration is counted only once.

c) The scopes shall be correctly nested, that is, 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.

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).

©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

d) When an identifier i visible in a scope P is re-declared in some inner scope Q enclosed within P, then:

— If the i declared in P refers to a named data type or a type label and the i declared in Q does not refere to a named data type or a type label, then both the i declared in P and the i declared in Q are visible in Q.

— Otherwise; 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 wherever the defined data type is visible, unless this 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 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 ;    --"entity1.wr1" obeys rule (a): a is visible in local scope
    wr2: a <> b ;   --"entity1.wr2" obeys rule (b): b is visible from outer scope
  END_ENTITY;

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

  ENTITY d;
    attr1 : INTEGER;
    attr2 : enum;
  WHERE
    wr1: ODD(attr1);  --"d.wr1" obeys rule (d) ODD is visible anywhere
    wr2: attr2 <> e;  --"d.wr2" obeys rule (e) e is visible outside the scope
```

 ©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
  END_ENTITY;         --"d.wr2"        defined by the type enum

END_SCHEMA;
```

10.3 Explicit item rules

The following subclauses 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.

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 every scope where the defined data type in which the enumeration item is declared is visible, unless this outer scope contains a declaration of the same identifier for some other item.

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, 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 which are visible within the scope of an entity declaration:

— attribute (explicit, derived and inverse);

— rule label (unique and domain rules);

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

EXAMPLE 1 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 2 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 which are visible within the scope of a function declaration:

— constant;

— entity;

— enumeration;

— function;

— parameter;

— procedure;

 ©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

— type;

— type label;

— variable.

EXAMPLE 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 which are visible within the scope of a procedure declaration:

— constant;

— entity;

— enumeration;

— function;

— parameter;

— procedure;

— type;

— type label;

— variable.

10.3.9 Query expression

See 12.6.7 for the definition of the QUERY expression.

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

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.

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.

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 may be used by implementations or in a remark tag (see 7.1.6.3.

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 which are visible within the scope of a rule declaration:

— constant;

— entity;

— enumeration;

— function;

— procedure;

— rule label;

— type;

— variable.

EXAMPLE 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;
```

 ©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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 may be used by implementations or in a remark tag (see 7.1.6.3.

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 which are visible within the scope of a schema declaration:

— constant;

— entity;

— enumeration;

— function;

— procedure;

— rule;

— subtype constraint;

— type.

EXAMPLE 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;
```

©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

10.3.14 Subtype constraint

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

NOTE The subtype constraint identifier may be used by implementations or in a remark tag (see 7.1.6.3.

Scope : A subtype constraint extends the scope of the entity for which it is declared. This scope extension extends from the keyword SUBTYPE_CONSTRAINT to the keyword END_SUBTYPE_CONSTRAINT which terminates that subtype constraint declaration.

10.3.15 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, 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 which are visible within the scope of a type declaration:

— enumeration;

— rule label (domain rule);

10.3.16 Type label

Visibility : A type label is visible in the scope of

— the entity and all subtypes of that entity,

— the function, or

— the procedure

in which it is declared. It is implicitly declared by its first appearance in the scope. For functions and procedures, the first appearance shall be in the formal parameter specification, and 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 result type specification of the function.

10.3.17 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

 ©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

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. ROLESOF (see 15.20) for an independent entity instance will return an empty set. An entity data type that 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 that 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:

```
242 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:

```
336 use_clause = USE FROM schema_ref [ '(' named_type_or_rename
                 { ',' named_type_or_rename } ')' ] ';' .
259 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;

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

— 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:

```
281 reference_clause = REFERENCE FROM schema_ref [ '(' resource_or_rename
                       { ',' resource_or_rename } ')' ] ';' .
288 resource_or_rename = resource_ref [ AS rename_id ] .
289 resource_ref = constant_ref | entity_ref | function_ref | procedure_ref |
                   type_ref .
284 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 1 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 (that means, USE specifications may be chained between schemas).

EXAMPLE 2 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;
```


©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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 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.

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

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, including the extensible defined data types that this interfaced type may extend using the BASED_ON keyword, but excluding any of the selectable items if the interfaced type is a SELECT type, also excluding those selectable items of select types that this interfaced type may be based on;

— 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 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;

— all subtype constraints for the interfaced entity data type;

— any constants, defined data types, entity data types or functions used in the declaration of attributes of the interfaced entity data type;

 ©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

— 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.

11.4.7 Subtype constraint interfaces

When a subtype constraint is interfaced, nothing is implicitly interfaced.

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

The constraints specified in the interfaced subtype constraint are reformed upon interfacing so as not to allow any complex entity data type to be instantiated in the current schema that was not allowed in the foreign schema (see annex C).

12 Expression

Expressions are combinations of operators, operands and function calls which are evaluated to produce a value.

Syntax:

```
216 expression = simple_expression [ rel_op_extended simple_expression ] .
283 rel_op_extended = rel_op | IN | LIKE .
282 rel_op = ’<’ | ’>’ | ’<=’ | ’>=’ | ’<>’ | ’=’ | ’:<>:’ | ’:=:’ .
305 simple_expression = term { add_like_op term } .
325 term = factor { multiplication_like_op factor } .
217 factor = simple_factor [ ’**’ simple_factor ] .
306 simple_factor = aggregate_initializer | entity_constructor |
                    enumeration_reference | interval | query_expression |
                    ( [ unary_op ] ( ’(’ expression ’)’ | primary ) ) .
331 unary_op = ’+’ | ’-’ | NOT .
269 primary = literal | ( qualifiable_factor { qualifier } ) .
257 multiplication_like_op = ’*’ | ’/’ | DIV | MOD | AND | ’||’ .
168 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.

©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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.

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.

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 $-10 * *2$ 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; thus, any fractional part is lost. If a and b are integers, it is always true that (a DIV b)*b + c*(a MOD b) = a, where c=1 for b>=0 and c=-1 for b<0. The absolute value of a MOD b shall be less than the absolute value of b. 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 (?).

©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Real number rounding

Rounding, when necessary, shall be determined from the precision p (either stated explicitly for 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 This example shows the effect of defining the number of significant digits in the fraction part of a real number, that is, 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 LOGI-

©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

CAL 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, and binary operands. These operators may also be applied to enumeration items declared in enumerations that are not extensible and that are not based on extensible enumerations. In addition, `=` and `<>` may be applied to values of aggregate and entity data types and enumeration items declared in extensible enumerations or enumerations based on extensible enumerations. 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 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, and so on, until an unequal pair is found or until all bit pairs have been examined. If an unequal pair is found, the

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

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, and so on, 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/IEC 10646 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 in an enumeration that is not extensible and that is not based on an extensible enumeration is based on their relative positions in the declaration of the enumeration data type. See rule (d) in 8.4.1.

For values whose data type is an extensible enumeration type or an enumeration type based on an extensible enumeration type, only comparison for equal or unequal is defined. Two such values are equal if they represent the same enumeration item, and unequal otherwise.

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, that is, `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;

 ©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

— 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.

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 such that both `s(l)` and `s(r)` produce either entity type values or are declared to be select data types, and if `TYPEOF(s(l))<>TYPEOF(s(r))`, then `l = r` is FALSE.

e) If there is a reference path s such that `NOT EXISTS(s(l))` or `NOT EXISTS(s(r))`, `l = r` is UNKNOWN.

f) Otherwise, `l = r` is TRUE.

EXAMPLE 1 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.

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

b) If the instances are the same instance, that is, `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, this is, `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 2 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));
```

 ©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
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, that is, `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;

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

— 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 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; that is, 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 All children have mothers but some children may have 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 entity instance
              -- 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;
```


©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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;
  (*"my_in" 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 ...
```


Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

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 that has a defined ordering, that is, the simple types (see 8.1), and defined data types whose underlying types are either simple types or enumeration types.

Syntax:

```
243 interval = '{' interval_low interval_op interval_item interval_op
              interval_high '}' .
246 interval_low = simple_expression .
247 interval_op = '<' | '<=' .
245 interval_item = simple_expression .
244 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 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.

 ©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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 1 To match the character `@`, the escape sequence `\@` is used.

The following examples illustrate these pattern matching characters

EXAMPLE 2 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
```

EXAMPLE 3 If `a := 'The quick red fox';`, the following holds:

```
a LIKE '$$$$'   --> TRUE
```

EXAMPLE 4 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

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)

NOTE The negation character (`!`) may be used before any character, not just the pattern matching characters, to match any character other than the negated character.

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

of 1 indicates the leftmost bit.

Syntax:

```
239 index_qualifier = '[' index_1 [ ':' index_2 ] ']' .
237 index_1 = index .
236 index = numeric_expression .
238 index_2 = index .
```

Rules and restrictions:

a) `index_1` shall evaluate to a positive integer or indeterminate (?) value.

b) $1 \leq$ `index_1` $\leq$ `BLENGTH(` binary value `)`, otherwise indeterminate (?) is returned.

c) `index_2`, when specified, shall evaluate to positive integer or indeterminate (?) value.

d) The `index_1` $\leq$ `index_2` $\leq$ `BLENGTH(`binary value`)`, otherwise indeterminate (?) is returned.

e) If either `index_1` or `index_2` evaluate to an indeterminate (?) value, indeterminate (?) is returned.

f) If the expression being indexed evaluates to indeterminate (?), indeterminate (?) is returned.

EXAMPLE 1 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 2 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. If either of the operands evaluates to indeterminate (?), the expression evaluates to indeterminate (?).

EXAMPLE 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. If either of the operands evaluates to indeterminate (?), that operand is dealt with as if it were the logical value UNKNOWN.

 ©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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.

Table 12 – NOT operator

Operand Value	Result Value
TRUE	FALSE
UNKNOWN	UNKNOWN
FALSE	TRUE

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.

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

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.

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

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.

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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:

```
239 index_qualifier = '[' index_1 [ ':' index_2 ] ']' .
237 index_1 = index .
236 index = numeric_expression .
238 index_2 = index .
```

Rules and restrictions:

a) index_1 shall evaluate to a positive integer or indeterminate (indeterminate (?)) value.

b) $1 \leq$ index_1 $\leq$ LENGTH(string value), otherwise indeterminate (?) is returned.

c) index_2, when specified, shall evaluate to positive integer or indeterminate (?) value.

d) The index_1 $\leq$ index_2 $\leq$ LENGTH(string value), otherwise indeterminate (?) is returned.

e) If either index_1 or index_2 evaluate to an indeterminate (?) value, indeterminate (?) is returned.

f) If the expression being indexed evaluates to indeterminate (?), indeterminate (?) is returned.

EXAMPLE 1 The seventh character of a string called name could be examined by

```
IF name[7]="00125FE1" THEN ...   -- assuming ISO 10646 representation
IF name[7:7]="00125FE1" THEN ... -- equivalent expression
```

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

©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

EXAMPLE 2 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 value, and the expression evaluates to a string value containing the concatenation of the two operands with the first operand appearing on the left. If either of the operands evaluates to indeterminate (?), the expression evaluates to indeterminate (?).

EXAMPLE 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.

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:

```
239 index_qualifier = '[' index_1 [ ':' index_2 ] ']' .
237 index_1 = index .
236 index = numeric_expression .
238 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.

b) The `index_1` shall evaluate to an integer value.

c) `LOINDEX(` aggregate value `)` $\leq$ `index_1` $\leq$ `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`.

©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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.

g) If either index_1 or index_2 evaluate to an indeterminate (?) value, indeterminate (?) is returned.

h) If the expression being indexed evaluates to indeterminate (?), indeterminate (?) is returned.

EXAMPLE Indexing for bags and sets may be used to iterate over all of the values in the aggregate value.

```
FUNCTION set_product(a_set : SET OF INTEGER):INTEGER;
LOCAL
  result: INTEGER := 1;
END_LOCAL;
  REPEAT index := LOINDEX(a_set) TO HIINDEX(a_set);
    result := result * a_set[index];
  END_REPEAT;

  RETURN(result);
END_FUNCTION;
```

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. If either of the operands evaluates to indeterminate (?), the expression evaluates to indeterminate (?).

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

 ©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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 are given in Table 17. The union operation is determined by the first condition in the following that holds:

a) If the left-hand operand is a bag value and the right-hand operand is a bag, list or set value whose elements are compatible with the base type of the bag, the result is the left-hand operand plus all the elements of the right-hand operand.

b) If the left-hand operand is a set value and the right-hand operand is a bag, list or set value whose elements are compatible with the base type of the set, 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.

c) If both operands are compatible lists, the resulting list is the left-hand operand, with the right-hand operand appended to the end.

   NOTE 1 The resulting list may contain duplicate elements, even if the operands are both declared as `LIST OF UNIQUE`.

d) 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 2 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.

e) If either of the operands evaluates to indeterminate (?), the expression evaluates to indeterminate (?).

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. That is, for each element of the second operand that 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 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

©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

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]

NOTE 1 First element of the second list follows the last element of the first list.
NOTE 2 New element becomes first in resultant list.
NOTE 3 New element becomes last in resultant list.

of the resulting aggregate value. If either of the operands evaluates to indeterminate (?), the expression evaluates to indeterminate (?).

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 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).

 ©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Table 19 – Subset and superset operators - operand types

First operand	Second operand
Bag	Bag
Bag	Set
Set	Bag
Set	Set

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` 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:

```
277 query_expression = QUERY '(' variable_id '<*' aggregate_source '|'
                       logical_expression ')' .
170 aggregate_source = simple_expression .
254 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) If the `aggregate_source` evaluates to indeterminate (`?`), the expression returns indeterminate (`?`).

d) 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`

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

evaluates to TRUE the element is added to the result; otherwise, it is not. If the `logical_expression` evaluates to indeterminate (?), that element is not a member of the resulting aggregate. 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.

NOTE If the aggregate source is an empty aggregation then that empty aggregation is the result.

EXAMPLE 1 Assuming `colour` is a defined type which has as its 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 2 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.

 ©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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 9.2.6), or as a local variable in a global rule (see 9.6).

EXAMPLE 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 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);
```


Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

```
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:

```
179 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 expression to the left of the attribute reference evaluates to indeterminate (?), the attribute reference expression evaluates to indeterminate (?). If the expression to the left of the attribute reference evaluates to a partial complex entity value then the attribute name to the right of the attribute reference shall occur within the entity declaration for that partial complex entity data type. If the declared type of the expression to the left of the attribute reference is an entity data type then the attribute name to the right of the attribute reference shall be declared in that entity data type, a supertype or subtype of that entity data type. If the declared type of the expression to the left of the attribute reference is a select data type, then the attribute name specified to the right shall be declared in an entity named in the select list or within a supertype or subtype of an entity named within the select list If the specified attribute is not present in the entity instance or partial complex entity value, indeterminate (?) is returned. If two or more attributes are found with the same name then the reference is ambiguous and indeterminate (?) is returned.

NOTE In the situation where ambiguous may occur it is recommended that the group reference qualifier be used to limit the scope of the reference.

EXAMPLE 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);
  second  : coloured_point := point(1.0,2.0,3.0)||coloured_point(red);
  x_coord : REAL;
END_LOCAL;
...
x_coord := first.x; --"foo" the value 1.0
IF first.colour = red THEN  (*"foo" colour is a valid reference since that
```


©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
                              attribute is present in the subtype
                              coloured_point, however, in this case the
                              attribute reference will return
                              indeterminate (?) since it is not present
                              in this instance. *)
IF second.colour = red THEN --"foo" 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:

```
232 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 expression to the left of the group reference evaluates to indeterminate (?), the group reference expression evaluates to indeterminate (?). If the declared type of the expression to the left of the group reference is an entity data type then the entity name specified to the right of the group reference shall be an entity in the same subtype/supertype graph as that entity data type. If the declared type of the expression to the left of the group reference is a select data type, then the entity name specified to the right shall occur in the select list or be an entity in the same subtype/supertype graph of an entity data named within the select list. 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 (||).

EXAMPLE 1 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;
```


Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

```
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)
*)
```

EXAMPLE 2 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;

ENTITY t;
  attr : BINARY;
END_ENTITY;

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

 ©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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.

NOTE The actual parameters for a function may evaluate to indeterminate (?). The function should correctly handle such values, and may itself return an indeterminate value.

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 (that is, as the value of a derived attribute or constant), the instance is considered as a part of the population under study.

Syntax:

```
219 function_call = ( built_in_function | function_ref ) [ actual_parameter_list ] .
167 actual_parameter_list = '(' parameter { ',' parameter } ')' .
264 parameter = expression .
```

Rules and restrictions:

The actual parameters passed shall be assignment compatible with the formal parameters.

EXAMPLE 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

An aggregate initializer is used to establish a value of type AGGREGATE OF GENERIC that may be assigned to 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).

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Syntax:

```
169 aggregate_initializer = '[' [ element { ',' element } ] ']' .
203 element = expression [ ':' repetition ] .
287 repetition = numeric_expression .
```

EXAMPLE 1 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. If the repetition evaluates to indeterminate (?) then the element expression to the left of the colon is not initialized into the aggregate.

EXAMPLE 2 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, that is, 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. If either of the operands evaluates to indeterminate (?), the expression evaluates to indeterminate (?). See annex B for further information on complex entity instances.

If an entity instance or a part of an entity instance (via a group reference) is used as the operand of a complex entity constructor then a shallow copy (see 9.2.6) of that instance or its attributes are used.

EXAMPLE Given:

```
ENTITY a
ABSTRACT SUPERTYPE;
  a1 : INTEGER;
END_ENTITY;
```

 ©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
  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
```

NOTE The first assignment to v1 copies the instance created by the complex instance construction operator, this instance contains the value of v2, not the v2 instance.

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 and constructed data types that are based on extensible data types);

— 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 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;
```


Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

```
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

12.12 Select data types in expressions

When verifying a schema a level 2 parser shall identify type compatibility of operands and operators in expressions. An expression involving a SELECT data type may only be valid for certain data specified in the select list, and not valid for other data types in the select list. So far, clause 12 has identified the valid data types in expressions ignoring the SELECT data types; the following rules are specifically for for these data types.

The data type returned by an expression containing operands whose declared type is a select data type is a select data type which contains all the return types from valid expressions possible from the specified operands.

The non-select data types of a select data type are the non-select data types of each data type in the select list of the select; the non-select data type of a data type which is not a select data type is the data type itself.

12.12.1 Select data types in unary expressions

This subclause specifies the handling of select data types in expressions with a single operand. This covers the operators -, +, NOT and the QUERY expression.

a) if all of the non-select data types in the select list of the declared type of the operand are valid in the context of the expression, the expression is valid and should return a valid result;

b) if some but not all of the non-select data types in the select list of the declared type of the operand are valid in the context of the expression, the expression is valid but may produce errors if values from those types which are invalid are evaluated in the expression;

c) when none of the non-select data types in the select list of the declared type of the operand are valid in the context of the expression, the expression is invalid and will always return an invalid result.

12.12.2 Select data types in binary expressions

This subclause specifies the handling of select data types in expressions with two operands.

 ©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

a) if for each non-select data type in the select list of the declared type of the left operand there exists a valid expression with each non-select data type in the select list of the declared type of the right operand, then the expression is valid and should return a valid result;

b) if some but not all of the non-select data types in the select list of the declared type of the left operand are valid in the context of the expression and at least one one non-select data type of the right operand, the expression is valid but may produce errors if values from those types which are invalid are evaluated in the expression;

c) when none of the non-select data types in the select list of the declared type of the left operand are valid in the context of the expression and any of the non-select data types of the right operand, the expression is invalid and will always return an invalid result.

12.12.3 Select data types in ternary expressions

This subclause specifies the handling of select data types in expressions with three operands.

The only EXPRESS expression that contains three operands is the interval expression. This is dealt with in the context of select data types as if it were two separate expressions which are ANDed together.

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, these are, 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:

```
309 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:

```
260 null_stmt = ';' .
```

EXAMPLE The following is a potential use of the null statement.

```
IF a = 13 THEN
  ; -- this is a null statement.
```


Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

```
ELSE
  b := 5 ;
END_IF ;
```

13.2 Alias statement

The ALIAS statement provides a local renaming capability for qualified variables and parameters.

Syntax:

```
174 alias_stmt = ALIAS variable_id FOR general_ref { qualifier } ';' stmt { stmt }
                 END_ALIAS ';' .
228 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 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

13.3.1 Assignment statement

The assignment statement is used to assign an instance to a local variable or parameter. If the expression to the right of the assignment statement is an entity instance then the assignment statement assigns a reference to that entity instance to the local variable or parameter. After such an assignment changes to the local variable or parameter are reflected in the original instance. An assignment statement may also be used to copy values to a local variable or instance, when they are declared to be non-entity data types. The data type of the value assigned to the variable shall be assignment compatible with the variable or parameter.

NOTE 1 An assignment statement cannot be used to create a value copy of an instance in a local variable or parameter.

 ©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Syntax:

```
176 assignment_stmt = general_ref { qualifier } ':=' expression ';' .
228 general_ref = parameter_ref | variable_ref .
```

EXAMPLE The following are valid assignments.

```
LOCAL
  a, b : REAL ;
  p    : point;
END_LOCAL ;
  ...
  a   := 1.1 ;
  b   := 2.5 * a;
  p.x := b ;
```

13.3.2 Assignment compatibility

For a value to be assigned to a derived attribute, local variable, or parameter, two conditions must be met:

NOTE 1 In the following text the term 'variable' is used to mean one of derived attribute, local variable, or parameter.

a) the resultant data type of the expression to be assigned must be type compatible with the data type of the variable;

b) the resultant value evaluated from the expression must satisfy all constraints defined for the data type of the variable.

The data type of the expression to be assigned and the data type of the variable are said to be compatible if any of the following hold true:

— the types are the same;

— the expression evaluates to a type that is a subtype or specialization of the type declared for the variable being assigned to;

— the declared data type of the variable being assigned to is a defined data type whose fundamental data type is a select data type, and the expression evaluates to a value of a data type which is assignment compatible with one, or more, of the data types specified in the domain of the select data type (including items added to that domain by other select data types based on the select data 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 the aggregation data type.

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

— if the object being assigned to is qualified, the following hold for the different qualifiers:

Attribute qualified:

- The declared type of the expression to the left of the attribute reference shall be either an entity data type or a select data type defined using at least one entity data type. The attribute named to the right of the attribute reference shall occur in either the entity data type or an entity in the same subtype/supertype graph as that entity data type.
- If the expression to the left of the attribute reference evaluates to an instance that contains the specified attribute and that attribute has a value, the original value is replaced by the expression to the right of the assignment statement, unless the object being assigned to is further qualified, in which case the further qualification is used.
- If the expression to the left of the attribute reference evaluates to an instance that contains the specified attribute and that attribute has an indeterminate (`?`) value (due to it being optional or not yet set), the expression to the right of the assignment statement is given to that attribute, unless the object being assigned to is further qualified, in which case it is in error.

Group qualified:

- The declared type of the expression to the left of the group reference shall be either an entity data type or a select data type defined using at least one entity data type. The entity name specified to the right of the group reference shall be an entity in the same subtype/supertype graph as the entity data type.
- If the expression to the left of the group reference evaluates to an instance that contains the entity name specified to the right of the group reference, the original partial complex entity value is replaced by the expression to the right of the assignment statement, unless the object being assigned to is further qualified, in which case the further qualification is used.

Element qualified:

- The declared type of the expression to the left of the element qualifier shall be one of (ARRAY, BINARY, LIST or STRING) or a select type defined using one of the previous types. The `index_1` value shall evaluate to an integer.
- The expression to the left of the element qualifier shall be initialized, that is, have a value, before assignment can be made to its elements.
- If the expression to the left of the element qualifier evaluates to an ARRAY and `{LOINDEX(left) <= index_1 <= HIINDEX(left)}` then:

  a) if there is an element already in the array at that position, the expression to the right of the assignment statement replaces the original value in the array at that position, unless the object being assigned to is further qualified in which case the further qualification is used on the original element;

©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

b) if there is an indeterminate (?) value in the array at that position that is not further qualified, the expression to the right of the assignment statement is inserted into the array at that position.

— If the expression to the left of the element qualifier evaluates to a BINARY and {1 <= index_1 <= BLENGTH(left)} then the expression to the right of the assignment statement replaces the bit in the binary at that position.

— If the expression to the left of the element qualifier evaluates to a LIST and {1 <= index_1 <= SIZEOF(left)} then the expression to the right of the assignment statement replaces the element in the list at that position, unless the object being assigned to is qualified in which case the further qualification is used.

— If the expression to the left of the element qualifier evaluates to a STRING and {1 <= index_1 <= LENGTH(left)} then the expression to the right of the assignment statement replaces the character in the string at that position.

Range qualified:

— The declared type of the expression to the left of the range qualifier shall be one of (BINARY or STRING) or a select type defined using one of the previous types. Both index_1 and index_2 shall evaluate to integer values.

— The expression to the left of the element qualifier shall be initialized, that is, have a value, before assignment can be made to its elements.

— If the expression to the left of the element qualifier evaluates to a BINARY and {1 <= index_1 <= index_2} AND (index_2 <= BLENGTH(left)) then the expression to the right of the assignment statement replaces the elements originally between index_1 and index_2.

NOTE 2 If BLENGTH(right) <> (index_2 - index_1 + 1) then BLENGTH(left) will be changed by this assignment.

— If the expression to the left of the element qualifier evaluates to a STRING and {1 <= index_1 <= index_2} AND (index_2 <= LENGTH(left)) then the expression to the right of the assignment statement replaces the elements originally between index_1 and index_2.

NOTE 3 If LENGTH(right) <> (index_2 - index_1 + 1) then LENGTH(left) will be changed by this assignment.

— If the object being assigned to is qualified and is not dealt with in the above cases, it is in error.

If the expression evaluates, in a level 4 compliant parser, to a type that is a generalization of the type declared for the variable being assigned to, the assignment statement is said to be not invalid. This implies that there may be valid assignments using this assignment statement, when the actual values returned from the expression obey the rules specified above, however, unpredictable results will occur if the actual values returned by the expression are not compatible using the rules defined above.

©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Those partial complex entity instances that 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 case label that is value equal to the case selector is selected and the statements associated with that label are executed. Value equality comparisons that result in UNKNOWN or FALSE are not selected. At most, one case-action is executed. If the selector evaluates to indeterminate (?) then the OTHERWISE clause, if present, is executed. If a case label evaluates to indeterminate (?), value equality returns UNKNOWN and the statement shall not be 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:

```
191 case_stmt = CASE selector OF { case_action } [ OTHERWISE ':' stmt ]
                END_CASE ';' .
299 selector = expression .
189 case_action = case_label { ',' case_label } ':' stmt .
190 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 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 ;
```

 ©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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:

```
192 compound_stmt = BEGIN stmt { stmt } END ';' .
```

EXAMPLE 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 a REPEAT which according to the repeat control would be in an infinite loop may be terminated.

Syntax:

```
214 escape_stmt = ESCAPE ';' .
```

Rules and restrictions:

An ESCAPE statement shall only occur within the scope of a REPEAT statement.

EXAMPLE 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, after END_REPEAT
```

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 UNKNOWN or indeterminate (?), the statement following ELSE is executed if the ELSE clause is present. If the `logical_expression` evaluates to FALSE UNKNOWN or indeterminate (?), and the ELSE clause is omitted, control is passed to the next statement.

©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Syntax:

```
233 if_stmt = IF logical_expression THEN stmt { stmt } [ ELSE stmt { stmt } ]
              END_IF ';' .
254 logical_expression = expression .
```

EXAMPLE 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:

```
270 procedure_call_stmt = ( built_in_procedure | procedure_ref )
                          [ actual_parameter_list ] ';' .
167 actual_parameter_list = '(' parameter { ',' parameter } ')' .
264 parameter = expression .
```

Rules and restrictions:

The actual parameters passed shall be assignment-compatible with the formal parameters.

EXAMPLE 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:

```
286 repeat_stmt = REPEAT repeat_control ';' stmt { stmt } END_REPEAT ';' .
285 repeat_control = [ increment_control ] [ while_control ] [ until_control ] .
235 increment_control = variable_id ':=' bound_1 TO bound_2 [ BY increment ] .
183 bound_1 = numeric_expression .
184 bound_2 = numeric_expression .
234 increment = numeric_expression .
```

 ©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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, UNKNOWN or indeterminate (?), 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 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; that is, 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:

```
235 increment_control = variable_id ':=' bound_1 TO bound_2 [ BY increment ] .
183 bound_1 = numeric_expression .
184 bound_2 = numeric_expression .
234 increment = numeric_expression .
```

Rules and restrictions:

a) The `numeric_expressions` representing the `bound_1`, `bound_2` and `increment` shall evaluate to numeric values.

b) The `numeric_expressions` representing the bounds and the increment are evaluated once on entry to the REPEAT statement.

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

c) If any of the numeric_expressions 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, UNKNOWN or indeterminate (?), the body is not executed.

Syntax:

```
339 while_control = WHILE logical_expression .
254 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.

 ©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

If the UNTIL control is the only control present, at least one iteration shall always be executed.

Syntax:

```
335 until_control = UNTIL logical_expression .
254 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:

```
290 return_stmt = RETURN [ '(' expression ')' ] ';' .
```

Rules and restrictions:

The RETURN statement shall only occur in a PROCEDURE or FUNCTION.

EXAMPLE 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:

```
308 skip_stmt = SKIP ';' .
```

Rules and restrictions:

The SKIP statement shall only occur in the scope of the REPEAT statement.

EXAMPLE The SKIP statement passes control to the END_REPEAT statement which causes the UNTIL control to to be evaluated.

```
REPEAT UNTIL (a=1);
```


Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

```
  ...
  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.

©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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. All built-in functions will return an indeterminate (?) result if passed an indeterminate (?) parameter unless explicitly specified otherwise in the description of the function.

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 `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 : $-1.0 \leq \mathtt{V} \leq 1.0$

EXAMPLE `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.

©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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 `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 `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 Use of the BLENGTH function.

```
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.

 ©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Result : The cosine of V ($-1.0 \leq$ result ≤ 1.0).

EXAMPLE `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 `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 `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.

©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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.

 ©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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

©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

15.9.3 Standard representation

The standard representation for an integer number is `'7I'`. The standard representation for a real number is `'10.1E'`. 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 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 Usage of HIINDEX function on nested aggregate values.

```
LOCAL
```

 ©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
  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 Usage of the LENGTH function.

```
LOCAL
  n : NUMBER;
  x1 : STRING := 'abc';
  x2 : STRING := "0000795E00006238";
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 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)
```


Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

```
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 `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 `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 `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.

©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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 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 The NVL function is used to supply zero (0.0) as the value of Z when Z is indeterminate (?).

```
ENTITY unit_vector;
  x, y : REAL;
  z : OPTIONAL REAL;
WHERE
  x**2 + y**2 + NVL(z, 0.0)**2 = 1.0;
END_ENTITY;
```

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.

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

EXAMPLE `ODD ( 121 ) --> TRUE`

15.20 RolesOf - general function

```
FUNCTION ROLESOF ( V:GENERIC_ENTITY ) : 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 (that is, '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. If V is indeterminate (?) then an empty set is returned.

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 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;
  ...
```


©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
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 ≤ 1.0).

EXAMPLE `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 Use of the SIZEOF function.

```
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`.

©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Conditions : V ≥ 0.0

EXAMPLE `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 `TAN ( 0.0 ) --> 0.0`

15.25 TypeOf - general function

```
FUNCTION TYPEOF ( V:GENERIC ) : SET OF STRING;
```

The TYPEOF function returns a SET of STRING that contains the names of all the data types that the parameter is a member of. Except for the simple data types (BINARY, BOOLEAN, INTEGER, LOGICAL, NUMBER, REAL, and STRING) and the aggregation data types (ARRAY, BAG, LIST, and SET), these names are qualified by the name of the schema that contains the definition of the type.

NOTE 1 The primary purpose of this function is to check whether a given value (variable or attribute value) can be used for a certain purpose, for example, 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 that the value V is a member of. Such names are qualified by the name of the schema that 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 by using both the type name that V belongs to (by declaration) and, if these are different also the type name that the instance of V represents, including their schema names if the type names are named data types, and applying the following rules:

   NOTE 2 When the actual parameter to TypeOf was a formal parameter to some function being evaluated, the "type to which V belongs (by declaration)" is the type declared for the original actual parameter, or the result data type for the actual parameter expression as specified in clause 12, and not the type declared for any formal parameter for which it has been substituted.

   — If V is an aggregate value, the type name is just the name of the aggregation data type (ARRAY, BAG, LIST, SET), not anything else.

   — If V is an enumeration data type based on another enumeration data type, add the

 ©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

names of the enumeration data types reached by traversing the BASED_ON relationships from the current enumeration data type.

— If V is an extensible enumeration data type, recursively add the names of the enumeration data types which are extensions to V.

NOTE 3 The two last items in the list above are true for an extensible enumeration data type that is based on another enumeration data type.

— If V evaluates to indeterminate (?), an empty SET is returned.

b) Repeat until the SET 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 that are compatible subtypes of a common supertype, or specializations 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:

— for each SELECT data type that the current name appears in as a member of the select list, do the following:

i) add the name of the select data type to the list;

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

ii) if the select data type is based on another select data type, add the names of the select data types reached by traversing the BASED_ON relationships from the current select data type;

iii) if the select data type is an extensible select data type, recursively add the names of the select data types that are extensions to the current select data type.

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 4 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 1 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 2 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;
```

In this context, the following statement is TRUE:

```
TYPEOF (lst) = ['ANOTHER_SCHEMA.HISLIST', 'THIS_SCHEMA.MYLIST', 'LIST']
```

 ©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

15.26 UsedIn - general function

FUNCTION USEDIN (T:GENERIC_ENTITY ; R:STRING) : BAG OF GENERIC_ENTITY ;

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.

If either T or R are indeterminate (?), an empty BAG is returned.

EXAMPLE 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; ...

(*"example"  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
 circles : SET OF circle := [];  -- an empty set of circle
END_LOCAL;
REPEAT i := LOINDEX(point) TO HIINDEX(point);
  circles := circles +
             USEDIN(point[i],'THIS_SCHEMA.CIRCLE.CENTRE');
END_REPEAT;
WHERE R1 : SIZEOF(
             QUERY(                                  -- start query
               at_zero <* circles |                  -- get all points
```


Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
              (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 Some results from calling VALUE with different parameters.

```
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 that is assignment compatible 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 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 ...
```


©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS

Not for Resale

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 The following test ensures that each point in a set is at a different position (by definition they are distinct, that is, 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. Built-in procedures will not be performed if passed an indeterminate (?) parameter, unless explicitly specified otherwise in the description of the procedure.

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 : $0 \leq P \leq$ SIZEOF(L)

©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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 : $1 \leq$ P $\leq$ SIZEOF(L)

 ©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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 subclauses: 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 BASED_ON = 'based_on' .
13 BEGIN = 'begin' .
14 BINARY = 'binary' .
15 BLENGTH = 'blength' .
16 BOOLEAN = 'boolean' .
17 BY = 'by' .
18 CASE = 'case' .
19 CONSTANT = 'constant' .

20 CONST_E = 'const_e' .
21 COS = 'cos' .
22 DERIVE = 'derive' .
23 DIV = 'div' .
24 ELSE = 'else' .
```


Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

```
25 END = 'end' .
26 END_ALIAS = 'end_alias' .
27 END_CASE = 'end_case' .
28 END_CONSTANT = 'end_constant' .
29 END_ENTITY = 'end_entity' .

30 END_FUNCTION = 'end_function' .
31 END_IF = 'end_if' .
32 END_LOCAL = 'end_local' .
33 END_PROCEDURE = 'end_procedure' .
34 END_REPEAT = 'end_repeat' .
35 END_RULE = 'end_rule' .
36 END_SCHEMA = 'end_schema' .
37 END_SUBTYPE_CONSTRAINT = 'end_subtype_constraint' .
38 END_TYPE = 'end_type' .
39 ENTITY = 'entity' .

40 ENUMERATION = 'enumeration' .
41 ESCAPE = 'escape' .
42 EXISTS = 'exists' .
43 EXTENSIBLE = 'extensible' .
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 GENERIC_ENTITY = 'generic_entity' .
53 HIBOUND = 'hibound' .
54 HIINDEX = 'hiindex' .
55 IF = 'if' .
56 IN = 'in' .
57 INSERT = 'insert' .
58 INTEGER = 'integer' .
59 INVERSE = 'inverse' .

60 LENGTH = 'length' .
61 LIKE = 'like' .
62 LIST = 'list' .
63 LOBOUND = 'lobound' .
64 LOCAL = 'local' .
65 LOG = 'log' .
66 LOG10 = 'log10' .
67 LOG2 = 'log2' .
68 LOGICAL = 'logical' .
69 LOINDEX = 'loindex' .

70 MOD = 'mod' .
71 NOT = 'not' .
72 NUMBER = 'number' .
73 NVL = 'nvl' .
74 ODD = 'odd' .
75 OF = 'of' .
76 ONEOF = 'oneof' .
77 OPTIONAL = 'optional' .
78 OR = 'or' .
```

 ©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
 79 OTHERWISE = 'otherwise' .

 80 PI = 'pi' .
 81 PROCEDURE = 'procedure' .
 82 QUERY = 'query' .
 83 REAL = 'real' .
 84 REFERENCE = 'reference' .
 85 REMOVE = 'remove' .
 86 RENAMED = 'renamed' .
 87 REPEAT = 'repeat' .
 88 RETURN = 'return' .
 89 ROLESOF = 'rolesof' .

 90 RULE = 'rule' .
 91 SCHEMA = 'schema' .
 92 SELECT = 'select' .
 93 SELF = 'self' .
 94 SET = 'set' .
 95 SIN = 'sin' .
 96 SIZEOF = 'sizeof' .
 97 SKIP = 'skip' .
 98 SQRT = 'sqrt' .
 99 STRING = 'string' .

100 SUBTYPE = 'subtype' .
101 SUBTYPE_CONSTRAINT = 'subtype_constraint' .
102 SUPERTYPE = 'supertype' .
103 TAN = 'tan' .
104 THEN = 'then' .
105 TO = 'to' .
106 TOTAL_OVER = 'total_over' .
107 TRUE = 'true' .
108 TYPE = 'type' .
109 TYPEOF = 'typeof' .

110 UNIQUE = 'unique' .
111 UNKNOWN = 'unknown' .
112 UNTIL = 'until' .
113 USE = 'use' .
114 USEDIN = 'usedin' .
115 VALUE = 'value' .
116 VALUE_IN = 'value_in' .
117 VALUE_UNIQUE = 'value_unique' .
118 VAR = 'var' .
119 WHERE = 'where' .

120 WHILE = 'while' .
121 WITH = 'with' .
122 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.

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

©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
125 digits = digit { digit } .
126 encoded_character = octet octet octet octet .
127 hex_digit = digit | 'a' | 'b' | 'c' | 'd' | 'e' | 'f' .
128 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' .
129 lparen_then_not_lparen_star = '(' { '(' } not_lparen_star { not_lparen_star } .

130 not_lparen_star = not_paren_star | ')' .
131 not_paren_star = letter | digit | not_paren_star_special .
132 not_paren_star_quote_special = '!' | '"' | '#' | '$' | '%' | '&' | '+' | ',' |
                               '-' | '.' | '/' | ':' | ';' | '<' | '=' | '>' |
                               '?' | '@' | '[' | '\' | ']' | '^' | '_' | '`' |
                               '{' | '|' | '}' | '~' .
133 not_paren_star_special = not_paren_star_quote_special | '''' .
134 not_quote = not_paren_star_quote_special | letter | digit | '(' | ')' | '*' .
135 not_rparen_star = not_paren_star | '(' .
136 octet = hex_digit hex_digit .
137 special = not_paren_star_quote_special | '(' | ')' | '*' | '''' .
138 not_rparen_star_then_rparen = not_rparen_star { not_rparen_star } ')' { ')' } .
```

A.1.3 Lexical elements

The following rules specify how certain combinations of characters are interpreted as lexical elements within the language.

```
139 binary_literal = '%' bit { bit } .

140 encoded_string_literal = '"' encoded_character { encoded_character } '"' .
141 integer_literal = digits .
142 real_literal = integer_literal |
                ( digits '.' [ digits ] [ 'e' [ sign ] digits ] ) .
143 simple_id = letter { letter | digit | '_' } .
144 simple_string_literal = \q { ( \q \q ) | not_quote | \s | \x9 | \xA | \xD } \q .
```

A.1.4 Remarks

The following rules specify the syntax of remarks in EXPRESS.

```
145 embedded_remark = '(*' [ remark_tag ] { ( not_paren_star { not_paren_star } ) |
                  lparen_then_not_lparen_star | ( '*' { '*' } ) |
                  not_rparen_star_then_rparen | embedded_remark } '*)' .
146 remark = embedded_remark | tail_remark .
147 remark_tag = '"' remark_ref { '.' remark_ref } '"' .
148 remark_ref = attribute_ref | constant_ref | entity_ref | enumeration_ref |
             function_ref | parameter_ref | procedure_ref | rule_label_ref |
             rule_ref | schema_ref | subtype_constraint_ref | type_label_ref |
             type_ref | variable_ref .
149 tail_remark = '--' [ remark_tag ] { \a | \s | \x9 | \xA | \xD } \n .
```

A.1.5 Interpreted identifiers

The following rules represent identifiers that are known to have a particular meaning (this is, to be declared elsewhere as types or functions or others).

©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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 multi-pass 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.

```
150 attribute_ref = attribute_id .
151 constant_ref = constant_id .
152 entity_ref = entity_id .
153 enumeration_ref = enumeration_id .
154 function_ref = function_id .
155 parameter_ref = parameter_id .
156 procedure_ref = procedure_id .
157 rule_label_ref = rule_label_id .
158 rule_ref = rule_id .
159 schema_ref = schema_id .

160 subtype_constraint_ref = subtype_constraint_id .
161 type_label_ref = type_label_id .
162 type_ref = type_id .
163 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`.

```
164 abstract_entity_declaration = ABSTRACT .
165 abstract_supertype = ABSTRACT SUPERTYPE ';' .
166 abstract_supertype_declaration = ABSTRACT SUPERTYPE [ subtype_constraint ] .
167 actual_parameter_list = '(' parameter { ',' parameter } ')' .
168 add_like_op = '+' | '-' | OR | XOR .
169 aggregate_initializer = '[' [ element { ',' element } ] ']' .

170 aggregate_source = simple_expression .
171 aggregate_type = AGGREGATE [ ':' type_label ] OF parameter_type .
172 aggregation_types = array_type | bag_type | list_type | set_type .
173 algorithm_head = { declaration } [ constant_decl ] [ local_decl ] .
174 alias_stmt = ALIAS variable_id FOR general_ref { qualifier } ';' stmt { stmt }
             END_ALIAS ';' .
175 array_type = ARRAY bound_spec OF [ OPTIONAL ] [ UNIQUE ] instantiable_type .
176 assignment_stmt = general_ref { qualifier } ':=' expression ';' .
177 attribute_decl = attribute_id | redeclared_attribute .
178 attribute_id = simple_id .
179 attribute_qualifier = '.' attribute_ref .

180 bag_type = BAG [ bound_spec ] OF instantiable_type .
181 binary_type = BINARY [ width_spec ] .
182 boolean_type = BOOLEAN .
183 bound_1 = numeric_expression .
184 bound_2 = numeric_expression .
185 bound_spec = '[' bound_1 ':' bound_2 ']' .
186 built_in_constant = CONST_E | PI | SELF | '?' .
187 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 |
```


Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

```
                     VALUE_UNIQUE .
188 built_in_procedure = INSERT | REMOVE .
189 case_action = case_label { ',' case_label } ':' stmt .

190 case_label = expression .
191 case_stmt = CASE selector OF { case_action } [ OTHERWISE ':' stmt ]
             END_CASE ';' .
192 compound_stmt = BEGIN stmt { stmt } END ';' .
193 concrete_types = aggregation_types | simple_types | type_ref .
194 constant_body = constant_id ':' instantiable_type ':=' expression ';' .
195 constant_decl = CONSTANT constant_body { constant_body } END_CONSTANT ';' .
196 constant_factor = built_in_constant | constant_ref .
197 constant_id = simple_id .
198 constructed_types = enumeration_type | select_type .
199 declaration = entity_decl | function_decl | procedure_decl |
              subtype_constraint_decl | type_decl .

200 derived_attr = attribute_decl ':' parameter_type ':=' expression ';' .
201 derive_clause = DERIVE derived_attr { derived_attr } .
202 domain_rule = [ rule_label_id ':' ] expression .
203 element = expression [ ':' repetition ] .
204 entity_body = { explicit_attr } [ derive_clause ] [ inverse_clause ]
              [ unique_clause ] [ where_clause ] .
205 entity_constructor = entity_ref '(' [ expression { ',' expression } ] ')' .
206 entity_decl = entity_head entity_body END_ENTITY ';' .
207 entity_head = ENTITY entity_id subsuper ';' .
208 entity_id = simple_id .
209 enumeration_extension = BASED_ON type_ref [ WITH enumeration_items ] .

210 enumeration_id = simple_id .
211 enumeration_items = '(' enumeration_id { ',' enumeration_id } ')' .
212 enumeration_reference = [ type_ref '.' ] enumeration_ref .
213 enumeration_type = [ EXTENSIBLE ] ENUMERATION [ ( OF
                   enumeration_items ) | enumeration_extension ] .
214 escape_stmt = ESCAPE ';' .
215 explicit_attr = attribute_decl { ',' attribute_decl } ':' [ OPTIONAL ]
                parameter_type ';' .
216 expression = simple_expression [ rel_op_extended simple_expression ] .
217 factor = simple_factor [ '**' simple_factor ] .
218 formal_parameter = parameter_id { ',' parameter_id } ':' parameter_type .
219 function_call = ( built_in_function | function_ref ) [ actual_parameter_list ] .

220 function_decl = function_head algorithm_head stmt { stmt } END_FUNCTION ';' .
221 function_head = FUNCTION function_id [ '(' formal_parameter
                { ';' formal_parameter } ')' ] ':' parameter_type ';' .
222 function_id = simple_id .
223 generalized_types = aggregate_type | general_aggregation_types |
                    generic_entity_type | generic_type .
224 general_aggregation_types = general_array_type | general_bag_type |
                            general_list_type | general_set_type .
225 general_array_type = ARRAY [ bound_spec ] OF [ OPTIONAL ] [ UNIQUE ]
                     parameter_type .
226 general_bag_type = BAG [ bound_spec ] OF parameter_type .
227 general_list_type = LIST [ bound_spec ] OF [ UNIQUE ] parameter_type .
228 general_ref = parameter_ref | variable_ref .
229 general_set_type = SET [ bound_spec ] OF parameter_type .

230 generic_entity_type = GENERIC_ENTITY [ ':' type_label ] .
231 generic_type = GENERIC [ ':' type_label ] .
```

 ©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
232 group_qualifier = '\' entity_ref .
233 if_stmt = IF logical_expression THEN stmt { stmt } [ ELSE stmt { stmt } ]
          END_IF ';' .
234 increment = numeric_expression .
235 increment_control = variable_id ':=' bound_1 TO bound_2 [ BY increment ] .
236 index = numeric_expression .
237 index_1 = index .
238 index_2 = index .
239 index_qualifier = '[' index_1 [ ':' index_2 ] ']' .

240 instantiable_type = concrete_types | entity_ref .
241 integer_type = INTEGER .
242 interface_specification = reference_clause | use_clause .
243 interval = '{' interval_low interval_op interval_item interval_op
           interval_high '}' .
244 interval_high = simple_expression .
245 interval_item = simple_expression .
246 interval_low = simple_expression .
247 interval_op = '<' | '<=' .
248 inverse_attr = attribute_decl ':' [ ( SET | BAG ) [ bound_spec ] OF ] entity_ref
               FOR [ entity_ref '.' ] attribute_ref ';' .
249 inverse_clause = INVERSE inverse_attr { inverse_attr } .

250 list_type = LIST [ bound_spec ] OF [ UNIQUE ] instantiable_type .
251 literal = binary_literal | logical_literal | real_literal | string_literal .
252 local_decl = LOCAL local_variable { local_variable } END_LOCAL ';' .
253 local_variable = variable_id { ',' variable_id } ':' parameter_type
                 [ ':=' expression ] ';' .
254 logical_expression = expression .
255 logical_literal = FALSE | TRUE | UNKNOWN .
256 logical_type = LOGICAL .
257 multiplication_like_op = '*' | '/' | DIV | MOD | AND | '||' .
258 named_types = entity_ref | type_ref .
259 named_type_or_rename = named_types [ AS ( entity_id | type_id ) ] .

260 null_stmt = ';' .
261 number_type = NUMBER .
262 numeric_expression = simple_expression .
263 one_of = ONEOF '(' supertype_expression { ',' supertype_expression } ')' .
264 parameter = expression .
265 parameter_id = simple_id .
266 parameter_type = generalized_types | named_types | simple_types .
267 population = entity_ref .
268 precision_spec = numeric_expression .
269 primary = literal | ( qualifiable_factor { qualifier } ) .

270 procedure_call_stmt = ( built_in_procedure | procedure_ref )
                      [ actual_parameter_list ] ';' .
271 procedure_decl = procedure_head algorithm_head { stmt } END_PROCEDURE ';' .
272 procedure_head = PROCEDURE procedure_id [ '(' [ VAR ] formal_parameter
                 { ';' [ VAR ] formal_parameter } ')' ] ';' .
273 procedure_id = simple_id .
274 qualifiable_factor = attribute_ref | constant_factor | function_call |
                     general_ref | population .
275 qualified_attribute = SELF group_qualifier attribute_qualifier .
276 qualifier = attribute_qualifier | group_qualifier | index_qualifier .
277 query_expression = QUERY '(' variable_id '<*' aggregate_source '|'
                   logical_expression ')' .
278 real_type = REAL [ '(' precision_spec ')' ] .
```

©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
279 redeclared_attribute = qualified_attribute [ RENAMED attribute_id ] .

280 referenced_attribute = attribute_ref | qualified_attribute .
281 reference_clause = REFERENCE FROM schema_ref [ '(' resource_or_rename
                       { ',' resource_or_rename } ')' ] ';' .
282 rel_op = '<' | '>' | '<=' | '>=' | '<>' | '=' | ':<>:' | ':=:' .
283 rel_op_extended = rel_op | IN | LIKE .
284 rename_id = constant_id | entity_id | function_id | procedure_id | type_id .
285 repeat_control = [ increment_control ] [ while_control ] [ until_control ] .
286 repeat_stmt = REPEAT repeat_control ';' stmt { stmt } END_REPEAT ';' .
287 repetition = numeric_expression .
288 resource_or_rename = resource_ref [ AS rename_id ] .
289 resource_ref = constant_ref | entity_ref | function_ref | procedure_ref |
                   type_ref .

290 return_stmt = RETURN [ '(' expression ')' ] ';' .
291 rule_decl = rule_head algorithm_head { stmt } where_clause END_RULE ';' .
292 rule_head = RULE rule_id FOR '(' entity_ref { ',' entity_ref } ')' ';' .
293 rule_id = simple_id .
294 rule_label_id = simple_id .
295 schema_body = { interface_specification } [ constant_decl ]
                  { declaration | rule_decl } .
296 schema_decl = SCHEMA schema_id [ schema_version_id ] ';' schema_body
                  END_SCHEMA ';' .
297 schema_id = simple_id .
298 schema_version_id = string_literal .
299 selector = expression .

300 select_extension = BASED_ON type_ref [ WITH select_list ] .
301 select_list = '(' named_types { ',' named_types } ')' .
302 select_type = [ EXTENSIBLE [ GENERIC_ENTITY ] ] SELECT [ select_list
                  | select_extension ] .
303 set_type = SET [ bound_spec ] OF instantiable_type .
304 sign = '+' | '-' .
305 simple_expression = term { add_like_op term } .
306 simple_factor = aggregate_initializer | entity_constructor |
                    enumeration_reference | interval | query_expression |
                    ( [ unary_op ] ( '(' expression ')' | primary ) ) .
307 simple_types = binary_type | boolean_type | integer_type | logical_type |
                   number_type | real_type | string_type .
308 skip_stmt = SKIP ';' .
309 stmt = alias_stmt | assignment_stmt | case_stmt | compound_stmt | escape_stmt |
           if_stmt | null_stmt | procedure_call_stmt | repeat_stmt | return_stmt |
           skip_stmt .

310 string_literal = simple_string_literal | encoded_string_literal .
311 string_type = STRING [ width_spec ] .
312 subsuper = [ supertype_constraint ] [ subtype_declaration ] .
313 subtype_constraint = OF '(' supertype_expression ')' .
314 subtype_constraint_body = [ abstract_supertype ] [ total_over ]
                              [ supertype_expression ';' ] .
315 subtype_constraint_decl = subtype_constraint_head subtype_constraint_body
                              END_SUBTYPE_CONSTRAINT ';' .
316 subtype_constraint_head = SUBTYPE_CONSTRAINT subtype_constraint_id FOR
                              entity_ref ';' .
317 subtype_constraint_id = simple_id .
318 subtype_declaration = SUBTYPE OF '(' entity_ref { ',' entity_ref } ')' .
319 supertype_constraint = abstract_entity_declaration |
                           abstract_supertype_declaration | supertype_rule .
```


©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
320 supertype_expression = supertype_factor { ANDOR supertype_factor } .
321 supertype_factor = supertype_term { AND supertype_term } .
322 supertype_rule = SUPERTYPE subtype_constraint .
323 supertype_term = entity_ref | one_of | '(' supertype_expression ')' .
324 syntax = schema_decl { schema_decl } .
325 term = factor { multiplication_like_op factor } .
326 total_over = TOTAL_OVER '(' entity_ref { ',' entity_ref } ')' ';' .
327 type_decl = TYPE type_id '=' underlying_type ';' [ where_clause ] END_TYPE ';' .
328 type_id = simple_id .
329 type_label = type_label_id | type_label_ref .

330 type_label_id = simple_id .
331 unary_op = '+' | '-' | NOT .
332 underlying_type = concrete_types | constructed_types .
333 unique_clause = UNIQUE unique_rule ';' { unique_rule ';' } .
334 unique_rule = [ rule_label_id ':' ] referenced_attribute { ','
                  referenced_attribute } .
335 until_control = UNTIL logical_expression .
336 use_clause = USE FROM schema_ref [ '(' named_type_or_rename
                { ',' named_type_or_rename } ')' ] ';' .
337 variable_id = simple_id .
338 where_clause = WHERE domain_rule ';' { domain_rule ';' } .
339 while_control = WHILE logical_expression .

340 width = numeric_expression .
341 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                     | 187
 1 ABSTRACT                | 164 165 166
 2 ACOS                    | 187
 3 AGGREGATE               | 171
 4 ALIAS                   | 174
 5 AND                     | 257 321
 6 ANDOR                   | 320
 7 ARRAY                   | 175 225
 8 AS                      | 259 288
 9 ASIN                    | 187
10 ATAN                    | 187
11 BAG                     | 180 226 248
12 BASED_ON                | 209 300
13 BEGIN                   | 192
14 BINARY                  | 181
15 BLENGTH                 | 187
16 BOOLEAN                 | 182
17 BY                      | 235
18 CASE                    | 191
19 CONSTANT                | 195
20 CONST_E                 | 186
21 COS                     | 187
22 DERIVE                  | 201
23 DIV                     | 257
24 ELSE                    | 233
25 END                     | 192
26 END_ALIAS               | 174
```


Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

```
27 END_CASE                | 191
28 END_CONSTANT            | 195
29 END_ENTITY              | 206
30 END_FUNCTION            | 220
31 END_IF                  | 233
32 END_LOCAL               | 252
33 END_PROCEDURE           | 271
34 END_REPEAT              | 286
35 END_RULE                | 291
36 END_SCHEMA              | 296
37 END_SUBTYPE_CONSTRAINT  | 315
38 END_TYPE                | 327
39 ENTITY                  | 207
40 ENUMERATION             | 213
41 ESCAPE                  | 214
42 EXISTS                  | 187
43 EXTENSIBLE              | 213 302
44 EXP                     | 187
45 FALSE                   | 255
46 FIXED                   | 341
47 FOR                     | 174 248 292 316
48 FORMAT                  | 187
49 FROM                    | 281 336
50 FUNCTION                | 221
51 GENERIC                 | 231
52 GENERIC_ENTITY          | 230 302
53 HIBOUND                 | 187
54 HIINDEX                 | 187
55 IF                      | 233
56 IN                      | 283
57 INSERT                  | 188
58 INTEGER                 | 241
59 INVERSE                 | 249
60 LENGTH                  | 187
61 LIKE                    | 283
62 LIST                    | 227 250
63 LOBOUND                 | 187
64 LOCAL                   | 252
65 LOG                     | 187
66 LOG10                   | 187
67 LOG2                    | 187
68 LOGICAL                 | 256
69 LOINDEX                 | 187
70 MOD                     | 257
71 NOT                     | 331
72 NUMBER                  | 261
73 NVL                     | 187
74 ODD                     | 187
75 OF                      | 171 175 180 191 213 225 226 227 229 248 250
                             303 313 318
76 ONEOF                   | 263
77 OPTIONAL                | 175 215 225
78 OR                      | 168
79 OTHERWISE               | 191
80 PI                      | 186
81 PROCEDURE               | 272
82 QUERY                   | 277
83 REAL                    | 278
84 REFERENCE               | 281
```

 ©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
 85 REMOVE                      | 188
 86 RENAMED                     | 279
 87 REPEAT                      | 286
 88 RETURN                      | 290
 89 ROLESOF                     | 187
 90 RULE                        | 292
 91 SCHEMA                      | 296
 92 SELECT                      | 302
 93 SELF                        | 186 275
 94 SET                         | 229 248 303
 95 SIN                         | 187
 96 SIZEOF                      | 187
 97 SKIP                        | 308
 98 SQRT                        | 187
 99 STRING                      | 311
100 SUBTYPE                     | 318
101 SUBTYPE_CONSTRAINT          | 316
102 SUPERTYPE                   | 165 166 322
103 TAN                         | 187
104 THEN                        | 233
105 TO                          | 235
106 TOTAL_OVER                  | 326
107 TRUE                        | 255
108 TYPE                        | 327
109 TYPEOF                      | 187
110 UNIQUE                      | 175 225 227 250 333
111 UNKNOWN                     | 255
112 UNTIL                       | 335
113 USE                         | 336
114 USEDIN                      | 187
115 VALUE                       | 187
116 VALUE_IN                    | 187
117 VALUE_UNIQUE                | 187
118 VAR                         | 272
119 WHERE                       | 338
120 WHILE                       | 339
121 WITH                        | 209 300
122 XOR                         | 168
123 bit                         | 139
124 digit                       | 125 127 131 134 143
125 digits                      | 141 142
126 encoded_character           | 140
127 hex_digit                   | 136
128 letter                      | 131 134 143
129 lparen_then_not_lparen_star | 145
130 not_lparen_star             | 129
131 not_paren_star              | 130 135 145
132 not_paren_star_quote_special | 133 134 137
133 not_paren_star_special      | 131
134 not_quote                   | 144
135 not_rparen_star             | 138
136 octet                       | 126
137 special                     |
138 not_rparen_star_then_rparen | 145
139 binary_literal              | 251
140 encoded_string_literal      | 310
141 integer_literal             | 142
142 real_literal                | 251
143 simple_id                   | 178 197 208 210 222 265 273 293 294 297 317
```


Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
                              328 330 337
144 simple_string_literal    | 310
145 embedded_remark          | 145 146
146 remark                   |
147 remark_tag               | 145 149
148 remark_ref               | 147
149 tail_remark              | 146
150 attribute_ref            | 148 179 248 274 280
151 constant_ref             | 148 196 289
152 entity_ref               | 148 205 232 240 248 258 267 289 292 316 318 323 326
153 enumeration_ref          | 148 212
154 function_ref             | 148 219 289
155 parameter_ref            | 148 228
156 procedure_ref            | 148 270 289
157 rule_label_ref           | 148
158 rule_ref                 | 148
159 schema_ref               | 148 281 336
160 subtype_constraint_ref   | 148
161 type_label_ref           | 148 329
162 type_ref                 | 148 193 209 212 258 289 300
163 variable_ref             | 148 228
164 abstract_entity_declaration | 319
165 abstract_supertype       | 314
166 abstract_supertype_declaration | 319
167 actual_parameter_list    | 219 270
168 add_like_op              | 305
169 aggregate_initializer    | 306
170 aggregate_source         | 277
171 aggregate_type           | 223
172 aggregation_types        | 193
173 algorithm_head           | 220 271 291
174 alias_stmt               | 309
175 array_type               | 172
176 assignment_stmt          | 309
177 attribute_decl           | 200 215 248
178 attribute_id             | 150 177 279
179 attribute_qualifier      | 275 276
180 bag_type                 | 172
181 binary_type              | 307
182 boolean_type             | 307
183 bound_1                  | 185 235
184 bound_2                  | 185 235
185 bound_spec               | 175 180 225 226 227 229 248 250 303
186 built_in_constant        | 196
187 built_in_function        | 219
188 built_in_procedure       | 270
189 case_action              | 191
190 case_label               | 189
191 case_stmt                | 309
192 compound_stmt            | 309
193 concrete_types           | 240 332
194 constant_body            | 195
195 constant_decl            | 173 295
196 constant_factor          | 274
197 constant_id              | 151 194 284
198 constructed_types        | 332
199 declaration              | 173 295
200 derived_attr             | 201
201 derive_clause            | 204
```

 ©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
202 domain_rule              | 338
203 element                  | 169
204 entity_body              | 206
205 entity_constructor       | 306
206 entity_decl              | 199
207 entity_head              | 206
208 entity_id                | 152 207 259 284
209 enumeration_extension    | 213
210 enumeration_id           | 153 211
211 enumeration_items        | 209 213
212 enumeration_reference    | 306
213 enumeration_type         | 198
214 escape_stmt              | 309
215 explicit_attr            | 204
216 expression               | 176 190 194 200 202 203 205 253 254 264 290 299 306
217 factor                   | 325
218 formal_parameter         | 221 272
219 function_call            | 274
220 function_decl            | 199
221 function_head            | 220
222 function_id              | 154 221 284
223 generalized_types        | 266
224 general_aggregation_types | 223
225 general_array_type       | 224
226 general_bag_type         | 224
227 general_list_type        | 224
228 general_ref              | 174 176 274
229 general_set_type         | 224
230 generic_entity_type      | 223
231 generic_type             | 223
232 group_qualifier          | 275 276
233 if_stmt                  | 309
234 increment                | 235
235 increment_control        | 285
236 index                    | 237 238
237 index_1                  | 239
238 index_2                  | 239
239 index_qualifier          | 276
240 instantiable_type        | 175 180 194 250 303
241 integer_type             | 307
242 interface_specification  | 295
243 interval                 | 306
244 interval_high            | 243
245 interval_item            | 243
246 interval_low             | 243
247 interval_op              | 243
248 inverse_attr             | 249
249 inverse_clause           | 204
250 list_type                | 172
251 literal                  | 269
252 local_decl               | 173
253 local_variable           | 252
254 logical_expression       | 233 277 335 339
255 logical_literal          | 251
256 logical_type             | 307
257 multiplication_like_op   | 325
258 named_types              | 259 266 301
259 named_type_or_rename     | 336
260 null_stmt                | 309
```


Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

```
261 number_type              | 307
262 numeric_expression       | 183 184 234 236 268 287 340
263 one_of                   | 323
264 parameter                | 167
265 parameter_id             | 155 218
266 parameter_type           | 171 200 215 218 221 225 226 227 229 253
267 population               | 274
268 precision_spec           | 278
269 primary                  | 306
270 procedure_call_stmt      | 309
271 procedure_decl           | 199
272 procedure_head           | 271
273 procedure_id             | 156 272 284
274 qualifiable_factor       | 269
275 qualified_attribute      | 279 280
276 qualifier                | 174 176 269
277 query_expression         | 306
278 real_type                | 307
279 redeclared_attribute     | 177
280 referenced_attribute     | 334
281 reference_clause         | 242
282 rel_op                   | 283
283 rel_op_extended          | 216
284 rename_id                | 288
285 repeat_control           | 286
286 repeat_stmt              | 309
287 repetition               | 203
288 resource_or_rename       | 281
289 resource_ref             | 288
290 return_stmt              | 309
291 rule_decl                | 295
292 rule_head                | 291
293 rule_id                  | 158 292
294 rule_label_id            | 157 202 334
295 schema_body              | 296
296 schema_decl              | 324
297 schema_id                | 159 296
298 schema_version_id        | 296
299 selector                 | 191
300 select_extension         | 302
301 select_list              | 300 302
302 select_type              | 198
303 set_type                 | 172
304 sign                     | 142
305 simple_expression        | 170 216 244 245 246 262
306 simple_factor            | 217
307 simple_types             | 193 266
308 skip_stmt                | 309
309 stmt                     | 174 189 191 192 220 233 271 286 291
310 string_literal           | 251 298
311 string_type              | 307
312 subsuper                 | 207
313 subtype_constraint       | 166 322
314 subtype_constraint_body  | 315
315 subtype_constraint_decl  | 199
316 subtype_constraint_head  | 315
317 subtype_constraint_id    | 160 316
318 subtype_declaration      | 312
319 supertype_constraint     | 312
```


©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
320 supertype_expression     | 263 313 314 323
321 supertype_factor         | 320
322 supertype_rule           | 319
323 supertype_term           | 321
324 syntax                   |
325 term                     | 305
326 total_over               | 314
327 type_decl                | 199
328 type_id                  | 162 259 284 327
329 type_label               | 171 230 231
330 type_label_id            | 161 329
331 unary_op                 | 306
332 underlying_type          | 327
333 unique_clause            | 204
334 unique_rule              | 333
335 until_control            | 285
336 use_clause               | 242
337 variable_id              | 163 174 235 253 277
338 where_clause             | 204 291 327
339 while_control            | 285
340 width                    | 341
341 width_spec               | 181 311
```


Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

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 (such as even numbers and odd numbers) others do have some members in common (such as 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:

— $A\&A \equiv A$
that is, a particular entity data type will occur only once within a given partial complex entity data type;

— $A\&B \equiv B\&A$
that is, the grouping of partial complex entity data types is commutative;

— $A\&(B\&C) \equiv (A\&B)\&C \equiv A\&B\&C$
that is, 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:

 ©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

— $A + [B1, B2] \equiv [B1, B2] + A \equiv [A, B1, B2]$
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 partial 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 and subtype constraint operators

Using the formalism above, it is possible to rewrite the restrictions defined in EXPRESS supertype expressions and subtype constraints 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.

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

These reductions alone do not completely describe the full meaning of the supertype expression or subtype constraints, in particular the ONEOF and TOTAL_OVER clauses. 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, that is,

$\text{ONEOF}(A, B, \ldots) \longrightarrow [A, B, \ldots]$

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.

$A \text{ AND } B \longrightarrow [A\&B]$
$A \text{ AND ONEOF}(B1, B2) \longrightarrow A\&[B1, B2] = [A\&B1, A\&B2]$
$\text{ONEOF}(A1, A2) \text{ AND ONEOF}(B1, B2) \longrightarrow [A1, A2]\&[B1, B2] =$
$[A1\&B1, A1\&B2, A2\&B1, A2\&B2]$

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.

$A \text{ ANDOR } B \longrightarrow [A, B, A\&B]$
$A \text{ ANDOR ONEOF}(B1, B2) \longrightarrow [A, [B1, B2], A\&[B1, B2]] = [A, B1, B2, A\&B1, A\&B2]$
$\text{ONEOF}(A1, A2) \text{ ANDOR ONEOF}(B1, B2) \longrightarrow [[A1, A2], [B1, B2], [A1, A2]\&[B1, B2]] =$
$[A1, A2, B1, B2, A1\&B1, A1\&B2, A2\&B1, A2\&B2]$

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 9.2.5.5.

EXAMPLE The expression below evaluates as follows:
$A \text{ ANDOR } B \text{ AND } C \longrightarrow [A, [B\&C], A\&[B\&C]] = [A, B\&C, A\&B\&C]$

B.3 Interpreting the possible complex entity data types

The interpretation of the supertype expressions and subtype constraints with additional information available from the declared structure, allows the developer of an EXPRESS schema to determine the complex entity data types that 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:

 ©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

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 i in the subtype/supertype graph within which a supertype constraint is declared, construct a SUBTYPE_CONSTRAINT of the form:

```
SUBTYPE_CONSTRAINT i_superconstraint FOR i;
  <supertype_constraint> ;
END_SUBTYPE_CONSTRAINT;
```

replacing `<supertype_constraint>` with the supertype constraint declared in the entity. Consider this constraint as part of the schema for the purposes of this algorithm. Ignore the supertype expression in the entity declaration that was the basis for the subtype constraint for the purposes of this algorithm.

NOTE 2 This step converts supertype constraints declared in the entity to equivalent SUBTYPE_CONSTRAINT declarations.

c) For each supertype i in the subtype/supertype graph, identify all data types $j1$, $j2$, ... jk in the subtype/supertype graph that are defined as subtypes of i, but do not occur in any SUBTYPE_CONSTRAINT defined for i in the schema or generated in step (b), and construct a SUBTYPE_CONSTRAINT of the form:

```
SUBTYPE_CONSTRAINT i_othersubtypes FOR i;
  j1 ANDOR j2 ANDOR ... ANDOR jk;
END_SUBTYPE_CONSTRAINT;
```

Consider this constraint as part of the schema for the purposes of this algorithm.

d) For each supertype i in the subtype/supertype graph, identify all SUBTYPE_CONSTRAINT $sc1$, $sc2$, ... sck that have i in their FOR clause. At this point, the parts of subtype constraints that contain total coverage or abstract restrictions are ignored. Combine the subtype expressions sxi of these constraints into a single SUBTYPE_CONSTRAINT sti of the form: ($sx1$ ANDOR $sx2$ ANDOR $sx3$ ... ANDOR sxk).

e) For each supertype i in the subtype/supertype graph, generate the evaluated set that represents the constraints between its immediate subtypes by applying the reductions in B.2 and the identities in B.1 to the SUBTYPE_CONSTRAINT sti given by step (d) above. Combine i with the result using the & operator. If i is not defined as an ABSTRACT SUPERTYPE in its ENTITY declaration or in any SUBTYPE_CONSTRAINT of i, add i to the result using +. Call this set E_i .

f) 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 the reductions in B.2 and the identities in B.1.

   2) Recursively apply step (f1) to each s, expanding subtypes of s until leaf entities are reached (for which no E_s are available).

©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

NOTE 3 This recursion must terminate, since there are no cycles in the subtype/supertype graph.

g) Combine root sets. Create $R = \sum_r E_r \equiv E_{r1} + E_{r2} + \ldots$, that is, R is the union of the sets produced in step (f).

h) For each supertype s in R, for each total coverage subtype constraint t_1, t_2 , … t_k defined for s:

1) Define t to be: (t_1 ANDOR t_2 … ANDOR t_k).

2) For all immediate subtypes s_i of s not in t_1, t_2 , … t_k replace each occurrence of s_i in R by the expression derived from (s_i and t) using the definitions in B.2.2.

3) Reduce R according to the reductions in B.2 and the identities in B.1.

i) 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 that include both m and s.

2) Generate the evaluated set of supertype combinations permitted by m, $P_m = R/m/s1\&R/m/s2\&\ldots$, that is, combine the evaluated sets produced in step (i1) using &.

3) Generate the evaluated set of supertype combinations that may not include all the supertypes of m, $X_m = \sum_s R/m/s$, that is, union together the evaluated sets produced in step (i1).

4) Put $R = (R - X_m) + P_m$.

j) For each SUBTYPE_CONSTRAINT expression k (including those generated in steps (b) and (h)) of the form ONEOF(S_1, S_2, …), 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 $D_k = \sum_{i,j} D_k^{i,j}$, that is, D_k is the union of the sets computed in step (j1).

3) Put $R = R - (R/D_k)$.

k) For each SUBTYPE_CONSTRAINT expression k (including those generated in steps (b) and (h)) 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 that are disallowed by k, $D_k^i = R/i - R/(Q_k/i)$.

3) Set $D_k = \sum_i D_k^i$, that is, D_k is the union of the sets computed in step (k2).

 ©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

4) Put $R = R - D_k$.

l) The final evaluated set R is the evaluated set for the input subtype/supertype graph.

EXAMPLE 1 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;
END_ENTITY;

SUBTYPE_CONSTRAINT p_subs FOR p;
  TOTAL_OVER(m, f);
  ONEOF(m, f) AND ONEOF(c, a);
END_SUBTYPE_CONSTRAINT;

ENTITY m SUBTYPE OF (p);
END_ENTITY;

ENTITY f SUBTYPE OF (p);
END_ENTITY;

ENTITY c SUBTYPE OF (p);
END_ENTITY;

ENTITY a SUBTYPE OF (p);
END_ENTITY;

SUBTYPE_CONSTRAINT no_li FOR a;
  ABSTRACT SUPERTYPE;
  ONEOF(l, i);
END_SUBTYPE_CONSTRAINT;

ENTITY l SUBTYPE OF (a);
END_ENTITY;

ENTITY i SUBTYPE OF (a);
END_ENTITY;

END_SCHEMA;
```

This schema is represented by EXPRESS-G in figure B.1.

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), (b), and (c).

— Applying step (d) gives:

```
SUBTYPE_CONSTRAINT stp FOR p;
TOTAL_OVER(m, f);
( (ONEOF(m, f) AND ONEOF(c, a)) );
END_SUBTYPE_CONSTRAINT;

SUBTYPE_CONSTRAINT sta FOR a;
(ONEOF(i,l));
END_SUBTYPE_CONSTRAINT;
```


Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

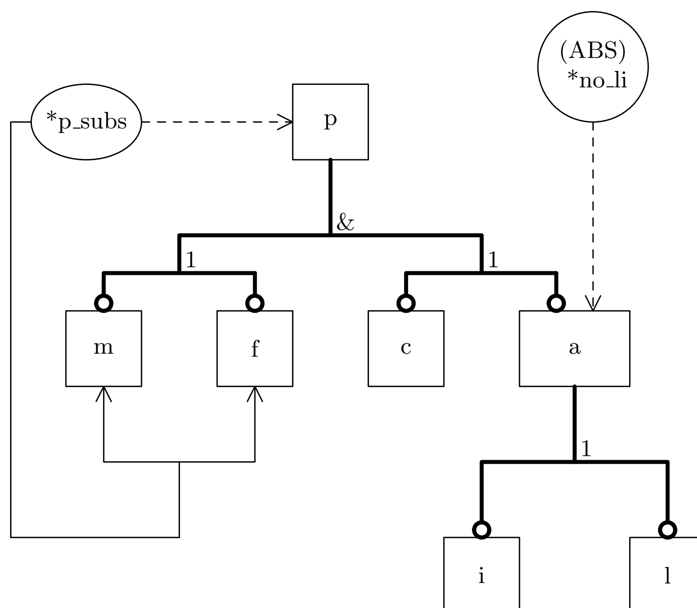


Figure B.1 – EXPRESS-G diagram of schema for example 1 on page 171

— Applying step (e) gives:

$$\begin{aligned} E_p &\longrightarrow [p\&m\&c, p\&m\&a, p\&f\&c, p\&f\&a, p] \\ E_a &\longrightarrow [a\&l, a\&i] \end{aligned}$$

— Applying step (f) expands the declarations of the root entities, in this case 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 (g) gives:

$$R = [p\&m\&c, p\&m\&a\&l, p\&m\&a\&i, p\&f\&c, p\&f\&a\&l, p\&f\&a\&i, p]$$

— Applying step (h) to the TOTAL_OVER constraint gives:

— TOTAL_OVER(m, f):

$$s_p = [p\&m, p\&f]$$

Replacing all occurrences of p that do not include m or f gives:

$$R = [p\&m\&c, p\&m\&a\&l, p\&m\&a\&i, p\&f\&c, p\&f\&a\&l, p\&f\&a\&i, p\&m, p\&f]$$

— There are no multiply inheriting subtypes, so step (i) is not required.

— Applying step (j) to each ONEOF constraint gives:

— ONEOF(m, f):

$$\begin{aligned} D_1^{1,2} &= [m\&f] \\ D_1 &= [m\&f] \end{aligned}$$

Removing D_1 from R according to step (j3) leaves R unchanged. Thus, we are left with:

$$R = [p\&m\&c, p\&m\&a\&l, p\&m\&a\&i, p\&f\&c, p\&f\&a\&l, p\&f\&a\&i, p\&m, p\&f]$$

©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

— 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 (j3) leaves R unchanged. Thus, we are left with:

$$R = [p\&m\&c, p\&m\&a\&l, p\&m\&a\&i, p\&f\&c, p\&f\&a\&l, p\&f\&a\&i, p\&m, p\&f]$$

— 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 (j3) leaves R unchanged. Thus, we are left with:

$$R = [p\&m\&c, p\&m\&a\&l, p\&m\&a\&i, p\&f\&c, p\&f\&a\&l, p\&f\&a\&i, p\&m, p\&f]$$

— Applying step (k) to each AND constraint gives:

— ONEOF(m, f) AND ONEOF(c, a):

$$\begin{aligned} Q_1 &= [m\&c, m\&a, f\&c, f\&a] \\ D_1^m &= [p\&m] \\ D_1^f &= [p\&f] \\ D_1^c &= [] \\ D_1^a &= [] \\ D_1 &= [p\&m, p\&f] \end{aligned}$$

Removing D_1 from R according to step (k4) changes R to:

$$R = [p\&m\&c, p\&m\&a\&l, p\&m\&a\&i, p\&f\&c, p\&f\&a\&l, p\&f\&a\&i]$$

— According to step (l), the result is thus:

$$R = [p\&m\&c, p\&m\&a\&l, p\&m\&a\&i, p\&f\&c, p\&f\&a\&l, p\&f\&a\&i]$$

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` who is `male` and a `citizen`.

— A `person` who is a `male alien` and who is an `illegal_alien`.

— A `person` who . . . .

In addition the TOTAL_OVER constraint ensures that evaluated sets calculated by any other schema which extends this subtype/supertype graph shall also include either `male` or `female` for valid instances of `person`.

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

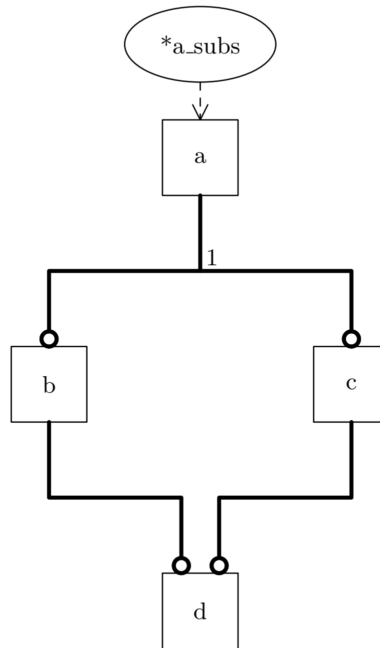


Figure B.2 – EXPRESS-G diagram of schema for example 2 on page 174

EXAMPLE 2 This example demonstrates that ONEOF is a global constraint that cannot be overridden by multiple inheritance.

```
SCHEMA diamond;

ENTITY a;
END_ENTITY;

SUBTYPE_CONSTRAINT a_subs FOR a;
  ONEOF(b, c);
END_SUBTYPE_CONSTRAINT;

ENTITY b SUBTYPE OF (a);
END_ENTITY;

ENTITY c SUBTYPE OF (a);
END_ENTITY;

ENTITY d SUBTYPE OF (b, c);
END_ENTITY;

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 required in steps (a) and (b).

— Step (c) creates:

```
SUBTYPE_CONSTRAINT b_othersubtypes FOR b;
  d;
```

 ©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

```
END_SUBTYPE_CONSTRAINT;

SUBTYPE_CONSTRAINT c_othersubtypes FOR c;
  d;
END_SUBTYPE_CONSTRAINT;
```

— Step (d) creates:

```
SUBTYPE_CONSTRAINT sca FOR a;
  ONEOF(b,c);
END_SUBTYPE_CONSTRAINT;

SUBTYPE_CONSTRAINT scb FOR b;
  d;
END_SUBTYPE_CONSTRAINT;

SUBTYPE_CONSTRAINT scc FOR c;
  d;
END_SUBTYPE_CONSTRAINT;
```

— Applying step (e) 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 (f) 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 (g) gives:

$$R = [a\&b\&d, a\&b, a\&c\&d, a\&c, a]$$

— Applying step (i) 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 (j) 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 (j3) removes the following elements from R: $[a\&b\&d\&c]$.

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Thus, we are left with:

$$R = [a\&b, a\&c, a]$$

— There are no supertype expressions that use AND, so step (k) is not required.

— According to step (l), the result is thus:

$$R = [a\&b, a\&c, a]$$

EXAMPLE 3 This example shows the effect of applying constraints to a complex structure that 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;
END_ENTITY;

ENTITY b SUBTYPE OF (a);
END_ENTITY;

ENTITY c SUBTYPE OF (a);
END_ENTITY;

ENTITY d SUBTYPE OF (a);
END_ENTITY;

ENTITY f SUBTYPE OF (a, z);
END_ENTITY;

ENTITY k SUBTYPE OF (d);
END_ENTITY;

ENTITY l SUBTYPE OF (d, y);
END_ENTITY;

ENTITY x SUBTYPE OF (z);
END_ENTITY;

ENTITY y SUBTYPE OF (z);
END_ENTITY;

ENTITY z;
END_ENTITY;

SUBTYPE_CONSTRAINT a_subs FOR a;
  ONEOF(b, c) AND d ANDOR f;
END_SUBTYPE_CONSTRAINT;

SUBTYPE_CONSTRAINT d_subs FOR d;
  ABSTRACT;
  ONEOF(k, l);
END_SUBTYPE_CONSTRAINT;

END_SCHEMA;
```

This is represented by EXPRESS-G in figure B.3.

The potential complex entity data types can be determined as follows:

©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

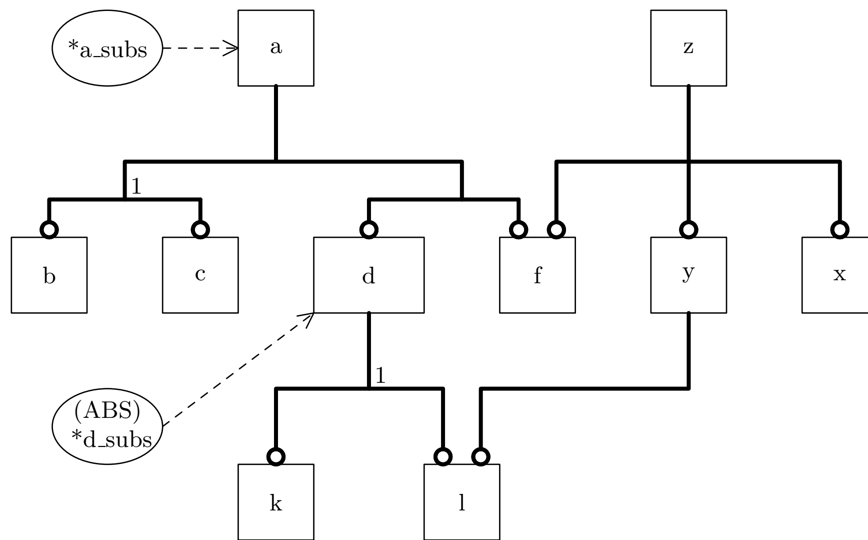


Figure B.3 – EXPRESS-G diagram of schema for example 3 on page 176

— The above EXPRESS explicitly specifies all of the entity declarations and complete supertype expressions as required in steps (a) and (b).

— Step (c) creates:

```
SUBTYPE_CONSTRAINT z_othersubtypes FOR z;
  f ANDOR y ANDOR x;
END_SUBTYPE_CONSTRAINT;

SUBTYPE_CONSTRAINT y_othersubtypes FOR y;
  l;
END_SUBTYPE_CONSTRAINT;
```

— Step (d) creates:

```
SUBTYPE_CONSTRAINT stz FOR z;
  f ANDOR y ANDOR x;
END_SUBTYPE_CONSTRAINT;

SUBTYPE_CONSTRAINT sty FOR y;
  l;
END_SUBTYPE_CONSTRAINT;

SUBTYPE_CONSTRAINT sta FOR a;
  ONEOF(b, c) AND d ANDOR f;
END_SUBTYPE_CONSTRAINT;

SUBTYPE_CONSTRAINT std FOR d;
  ABSTRACT;
  ONEOF(k, l);
END_SUBTYPE_CONSTRAINT;
```

— Applying step (e) gives:

$$E_a \longrightarrow [a, a\&b\&d, a\&b\&d\&f, a\&c\&d, a\&c\&d\&f, a\&f]$$
$$E_d \longrightarrow [d\&k, d\&l]$$
$$E_y \longrightarrow [l\&y, y]$$
$$E_z \longrightarrow [f\&x\&y\&z, f\&x\&z, f\&y\&z, f\&z, x\&y\&z, x\&z, y\&z, z]$$

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

— Applying step (f) 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 (g) gives:

$$R = [a, a\&b\&d\&k, a\&b\&d\&l, a\&b\&d\&f\&k, a\&b\&d\&f\&l, a\&c\&d\&k, a\&c\&d\&l, a\&c\&d\&f\&k, a\&c\&d\&f\&l, a\&f, f\&l\&x\&y\&z, f\&l\&y\&z, f\&x\&y\&z, f\&x\&z, f\&y\&z, f\&z, l\&x\&y\&z, l\&y\&z, x\&y\&z, x\&z, y\&z, z]$$

— There are no total coverage subtype constraints, so step (h) is not required.

— Applying step (i) 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:

$$C_l^d = [a\&b\&d\&f\&k\&l\&y\&z, a\&b\&d\&f\&k\&l\&x\&y\&z, a\&b\&d\&f\&l\&z, a\&b\&d\&f\&l\&x\&z, a\&b\&d\&f\&l\&y\&z, a\&b\&d\&f\&l\&x\&y\&z, a\&b\&d\&l, a\&c\&d\&f\&k\&l\&y\&z, a\&c\&d\&f\&k\&l\&x\&y\&z, a\&c\&d\&f\&l\&z, a\&c\&d\&f\&l\&x\&z, a\&c\&d\&f\&l\&y\&z, a\&c\&d\&f\&l\&x\&y\&z, a\&c\&d\&l]$$
$$C_l^y = [a\&b\&d\&f\&k\&l\&y\&z, a\&b\&d\&f\&k\&l\&x\&y\&z, a\&b\&d\&f\&l\&y\&z, a\&b\&d\&f\&l\&x\&y\&z, a\&c\&d\&f\&k\&l\&y\&z, a\&c\&d\&f\&k\&l\&x\&y\&z, a\&c\&d\&f\&l\&y\&z, a\&c\&d\&f\&l\&x\&y\&z, a\&f\&l\&y\&z, a\&f\&l\&x\&y\&z, l\&x\&y\&z\ l\&y\&z]$$

 ©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

$$
\begin{aligned}
P_l &= [a\&b\&c\&d\&f\&k\&l\&y\&z, a\&b\&c\&d\&f\&k\&l\&x\&y\&z, a\&b\&c\&f\&l\&y\&z, \\
&\quad a\&b\&c\&f\&l\&x\&y\&z, a\&b\&d\&f\&k\&l\&y\&z, a\&b\&d\&f\&k\&l\&x\&y\&z, \\
&\quad a\&b\&d\&f\&l\&y\&z, a\&b\&d\&f\&l\&x\&y\&z, a\&b\&d\&l\&x\&y\&z, a\&b\&d\&l\&y\&z, \\
&\quad a\&c\&d\&f\&k\&l\&y\&z, a\&c\&d\&f\&k\&l\&x\&y\&z, a\&c\&d\&f\&l\&y\&z, \\
&\quad a\&c\&d\&f\&l\&x\&y\&z, a\&c\&d\&l\&x\&y\&z, a\&c\&d\&l\&y\&z] \\
X_l &= [a\&b\&d\&f\&k\&l\&y\&z, a\&b\&d\&f\&k\&l\&x\&y\&z, a\&b\&d\&f\&l\&z, \\
&\quad a\&b\&d\&f\&l\&x\&z, a\&b\&d\&f\&l\&y\&z, a\&b\&d\&f\&l\&x\&y\&z, a\&b\&d\&l, \\
&\quad a\&c\&d\&f\&k\&l\&y\&z, a\&c\&d\&f\&k\&l\&x\&y\&z, a\&c\&d\&f\&l\&z, a\&c\&d\&f\&l\&x\&z, \\
&\quad a\&c\&d\&f\&l\&y\&z, a\&c\&d\&f\&l\&x\&y\&z, a\&c\&d\&l, a\&f\&l\&y\&z, a\&f\&l\&x\&y\&z, \\
&\quad l\&x\&y\&z, l\&y\&z]
\end{aligned}
$$

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 (j) 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 (j3) 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 (j3) 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:

$$
\begin{aligned}
R &= [a, a\&b\&d\&f\&k\&x\&z, a\&b\&d\&f\&k\&y\&z, a\&b\&d\&f\&k\&x\&y\&z, \\
&\quad a\&b\&d\&f\&k\&z, a\&b\&d\&f\&l\&y\&z, a\&b\&d\&f\&l\&x\&y\&z, a\&b\&d\&k,
\end{aligned}
$$

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

$a\&b\&d\&l\&x\&y\&z, a\&b\&d\&l\&y\&z, a\&c\&d\&f\&l\&y\&z, a\&c\&d\&f\&l\&x\&y\&z,$
$a\&c\&d\&f\&k\&x\&z, a\&c\&d\&f\&k\&y\&z, a\&c\&d\&f\&k\&x\&y\&z, a\&c\&d\&f\&k\&z,$
$a\&c\&d\&k, a\&c\&d\&l\&x\&y\&z, a\&c\&d\&l\&y\&z, a\&f\&z, a\&f\&x\&z, a\&f\&y\&z,$
$a\&f\&x\&y\&z, x\&y\&z, x\&z, y\&z, z]$

— Applying step (k) to each AND constraint gives:

— ONEOF(b, c) AND d:

$$\begin{aligned} Q_1 &= [b\&d, c\&d] \\ D_1^b &= [] \\ D_1^c &= [] \\ D_1^d &= [] \\ D_1 &= [] \end{aligned}$$

Removing D_1 from R according to step (k4) leaves R unchanged. Thus, we are left with:

$$\begin{aligned} R = [&a, a\&b\&d\&f\&k\&x\&z, a\&b\&d\&f\&k\&y\&z, a\&b\&d\&f\&k\&x\&y\&z, \\ &a\&b\&d\&f\&k\&z, a\&b\&d\&f\&l\&y\&z, a\&b\&d\&f\&l\&x\&y\&z, a\&b\&d\&k, \\ &a\&b\&d\&l\&x\&y\&z, a\&b\&d\&l\&y\&z, a\&c\&d\&f\&l\&y\&z, a\&c\&d\&f\&l\&x\&y\&z, \\ &a\&c\&d\&f\&k\&x\&z, a\&c\&d\&f\&k\&y\&z, a\&c\&d\&f\&k\&x\&y\&z, a\&c\&d\&f\&k\&z, \\ &a\&c\&d\&k, a\&c\&d\&l\&x\&y\&z, a\&c\&d\&l\&y\&z, a\&f\&z, a\&f\&x\&z, a\&f\&y\&z, \\ &a\&f\&x\&y\&z, x\&y\&z, x\&z, y\&z, z] \end{aligned}$$

— According to step (l), the result is thus:

$$\begin{aligned} R = [&a, a\&b\&d\&f\&k\&x\&z, a\&b\&d\&f\&k\&y\&z, a\&b\&d\&f\&k\&x\&y\&z, \\ &a\&b\&d\&f\&k\&z, a\&b\&d\&f\&l\&y\&z, a\&b\&d\&f\&l\&x\&y\&z, a\&b\&d\&k, \\ &a\&b\&d\&l\&x\&y\&z, a\&b\&d\&l\&y\&z, a\&c\&d\&f\&l\&y\&z, a\&c\&d\&f\&l\&x\&y\&z, \\ &a\&c\&d\&f\&k\&x\&z, a\&c\&d\&f\&k\&y\&z, a\&c\&d\&f\&k\&x\&y\&z, a\&c\&d\&f\&k\&z, \\ &a\&c\&d\&k, a\&c\&d\&l\&x\&y\&z, a\&c\&d\&l\&y\&z, a\&f\&z, a\&f\&x\&z, a\&f\&y\&z, \\ &a\&f\&x\&y\&z, x\&y\&z, x\&z, y\&z, z] \end{aligned}$$

©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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 annex. The following reductions are used to remove these holes from a supertype expression:

— ONEOF(*A*, <>, ...) ⟶ ONEOF(*A*, ...)

— ONEOF(<>) ⟶ <>

— ONEOF(*A*) ⟶ *A*

— *A* AND <> ⟶ ONEOF(*A*, *A*)

— *A* ANDOR <> ⟶ *A*

— TOTAL_OVER(*A*, <>, ...) ⟶ TOTAL_OVER(*A*, ...)

— TOTAL_OVER(<>) ⟶ <>

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 1 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

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

the holes thus introduced, producing a valid supertype expression referring only to entities in the entity pool.

c) Compute the evaluated set according to the algorithm in annex B starting with step (b) and applying the reductions specified in the first two paragraphs of this annex to the constraints generated in steps (b), (c), and (h) specified in B.3.

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 2 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.

EXAMPLE 1 The schema `example` (see example 1 on page 171 in annex B) is used to demonstrate the algorithm.

```
SCHEMA test;
USE FROM example (l);
REFERENCE FROM example (m, c);
END_SCHEMA;
```

The potential complex entity data types are determined as follows:

— The entity pool is l, m, c, a, p: l, m, and c 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:

$$\text{ONEOF}(m, f) \text{ AND } \text{ONEOF}(c, a)$$
$$\text{ONEOF}(m, <>) \text{ AND } \text{ONEOF}(c, a)$$
$$\text{ONEOF}(m) \text{ AND } \text{ONEOF}(c, a)$$
$$m \text{ AND } \text{ONEOF}(c, a)$$

Similarly, for a we find:

$$\text{ONEOF}(l, i)$$
$$\text{ONEOF}(l, <>)$$
$$\text{ONEOF}(l)$$
$$l$$

— In this case, the supertype expressions are already complete, as required by step (c).

— Applying the evaluated set algorithm in step (c) gives the evaluated set $R = [c\&m\&p, a\&l\&m\&p]$.

The complex entity data type $a\&l\&m\&p$ contains the explicitly USE'd entity l, and so can be independently instantiated. $c\&m\&p$, on the other hand, cannot be independently instantiated in the current schema.

EXAMPLE 2 Assuming the following schemas are available:

```
SCHEMA s1;
ENTITY e1 SUPERTYPE OF (e11 ANDOR e12); END_ENTITY;
```

 ©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
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).

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

— Applying the evaluated set algorithm in step (c) 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 $e1\&f\&e211$, $e1\&f\&g\&e211$, $e1\&g\&e322$, $e1\&f\&g\&e322$, and $e1\&f\&g\&e211\&e322$ contain one of the explicitly USE'd entity $e211$ or $e322$, and so can be independently instantiated. On the other hand, $e1$, $e1\&f$, $e1\&g$, and $e1\&f\&g$ cannot be independently instantiated in the current schema.

©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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.

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; there is no requirement to have an associated EXPRESS specification.

EXAMPLE Figure D.1 and Figure D.2 show an EXPRESS-G diagram for the single EXPRESS schema given in the example in J.1 on page 242 The graphical diagram is divided to illustrate the use of multiple pages.

The principal elements of the diagram show that 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 lines joining the boxes. Differing line styles provide information on the kind of definition or relationship.

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

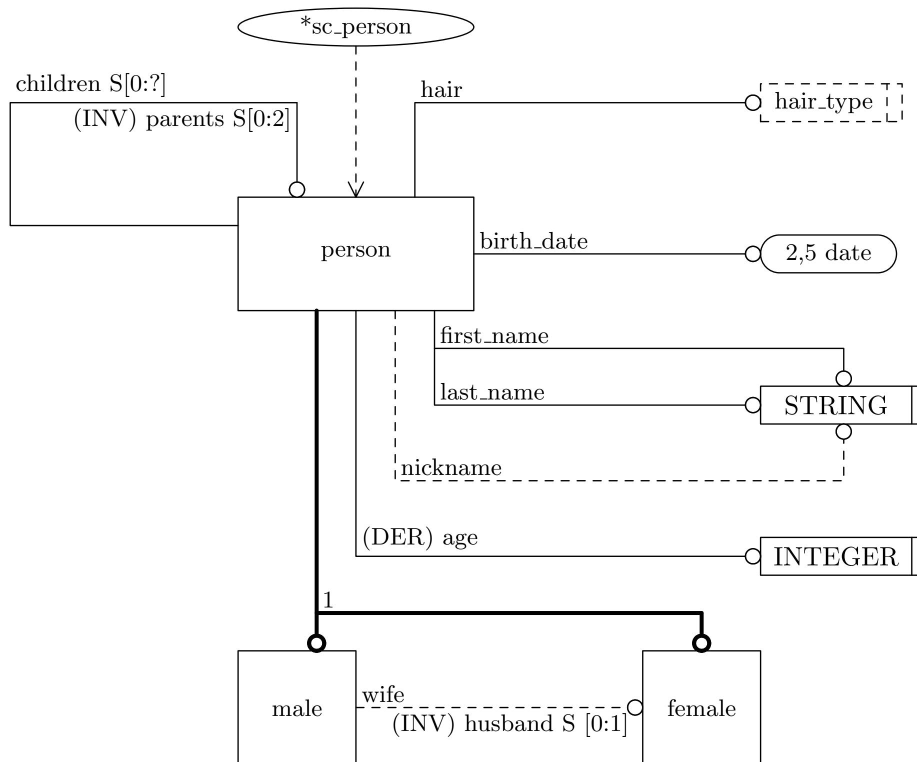


Figure D.1 – Complete entity level diagram of the example in J.1 on page 242 (Page 1 of 2)

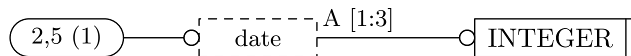


Figure D.2 – Complete entity level diagram of the example in J.1 on page 242 (Page 2 of 2)

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.

D.2.1.1 Symbols for generalized data types

The symbol for the EXPRESS GENERIC_ENTITY data type is the same as for EXPRESS simple data types. The name of the data type is enclosed within the box as shown in figure D.4.

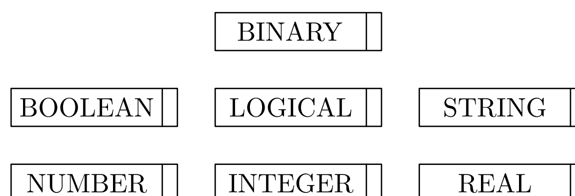


Figure D.3 – Symbols for EXPRESS simple data types

©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

GENERIC_ENTITY

Figure D.4 – Symbol for EXPRESS generic_entity data type

SELECT ENUMERATION

Figure D.5 – Symbols for EXPRESS constructed 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.5.

The symbol for a SELECT data type consists of a dashed box with a double vertical line at the left. If the select data type is a GENERIC_ENTITY SELECT data type, the data type name is preceded by a superscripted asterisk (*) symbol.

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.6 (see D.5.4).

EXAMPLE The two diagrams in Figure D.7 are equivalent:

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.

a_select an_enumeration

Figure D.6 – Abbreviated symbols for the EXPRESS constructed data types when used as the representation of defined data types

hair_type ENUMERATION

hair_type

Figure D.7 – Example of alternative methods for representing an enumeration data type

©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

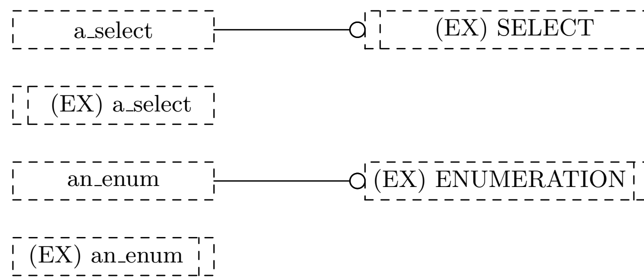


Figure D.8 – Symbols for EXPRESS extensible constructed data types

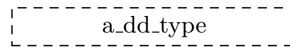


Figure D.9 – Symbol for EXPRESS defined data type

D.2.2.1 Extensible constructed data types

Extensible constructed data types are denoted in EXPRESS-G by placing the characters `EX` enclosed in parentheses, that is `(EX)`, before the name of the constructed data type as shown in figure D.8.

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.9.

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.10.

D.2.5 Symbols for subtype_constraints

The symbol for a SUBTYPE_CONSTRAINT is an ellipse enclosing the name of the SUBTYPE_CONSTRAINT, as shown in Figure D.11.

D.2.6 Symbols for functions and procedures

EXPRESS-G does not support any notation for either FUNCTION or PROCEDURE definitions.

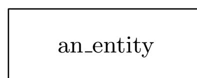


Figure D.10 – Symbol for an EXPRESS entity data type

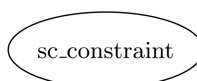


Figure D.11 – Symbol for an EXPRESS subtype_constraint

 ©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

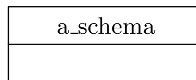


Figure D.12 – Symbol for a schema

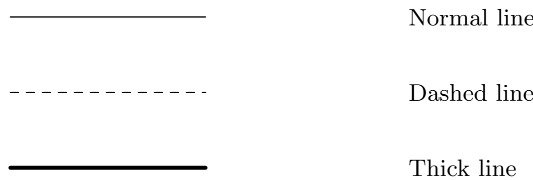


Figure D.13 – Relationship line styles

D.2.7 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).

D.2.8 Symbols for schemas

The symbol for a SCHEMA (Figure D.12) 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.

D.3 Relationship symbols

Definition symbols are connected by lines of various styles as shown in Figure D.13.

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. A dashed line shall also be drawn between the ellipse of the SUBTYPE_CONSTRAINT and the constrained supertype entity box An inheritance relationship (that is, a subtype and supertype relationship) is presented as a thick line. The extension of one constructed data type by another is also 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, that is, the circle is at the subtype end of the line. For the extension of constructed data types, the emphasized direction is toward the constructed data type that is based on the extensible data type (that is, the circle is at the end of the line that is attached to the constructed data type that is based on the extensible constructed data type).

EXAMPLE Relationship directions are illustrated in Figure D.14, which is an incomplete rendition of the EXPRESS code given in the example in J.2 on page 243. The diagram consists of six 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`

©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

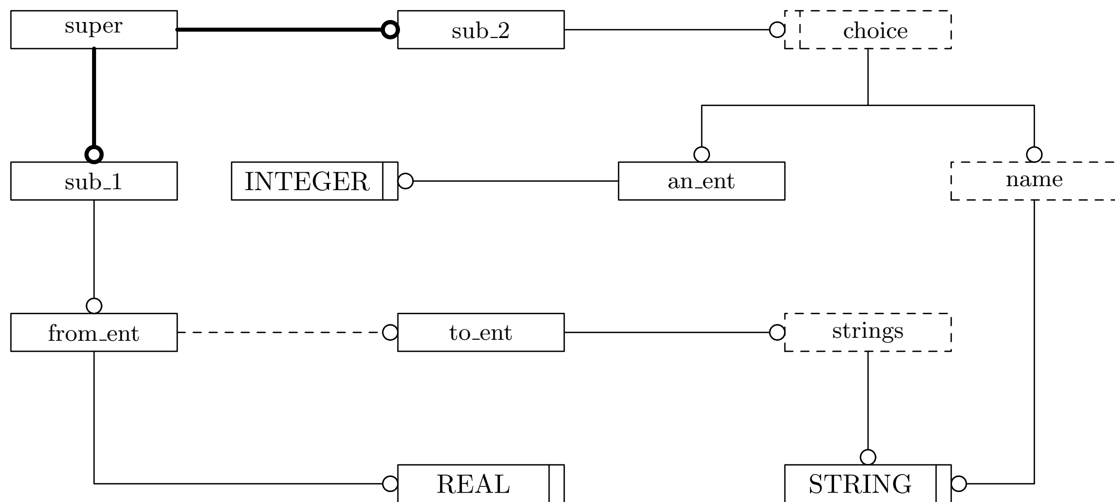


Figure D.14 – Partial entity level diagram illustrating relationship directions from the example in J.2 on page 243 (Page 1 of 1)

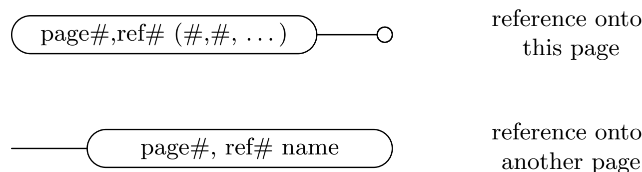


Figure D.15 – Composition symbols: page references

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.

NOTE 1 Although the example diagrams show only straight relationship lines, lines may follow any path (they may, for example, be curved).

NOTE 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.15.

A schema may reference definitions from another schema. The symbols for inter-schema references are shown in Figure D.16.

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.15. 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

 ©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

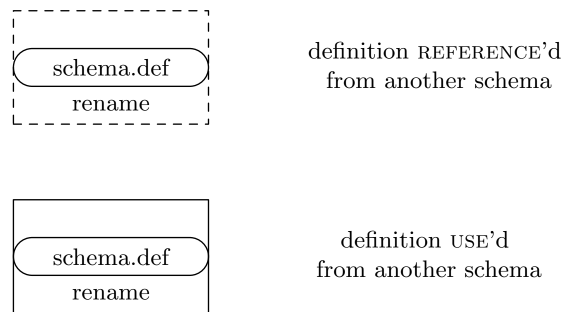


Figure D.16 – Composition symbols: inter-schema references

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 enclosed 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.16.

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.24.

D.5 Entity level diagrams

EXPRESS-G may be used to represent the definitions, and their relationships, within one schema. This type of diagram may consist of simple data types, defined data types, entity data types, subtype constraints, and relationship symbols, together with role and cardinality information as appropriate to 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.

©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

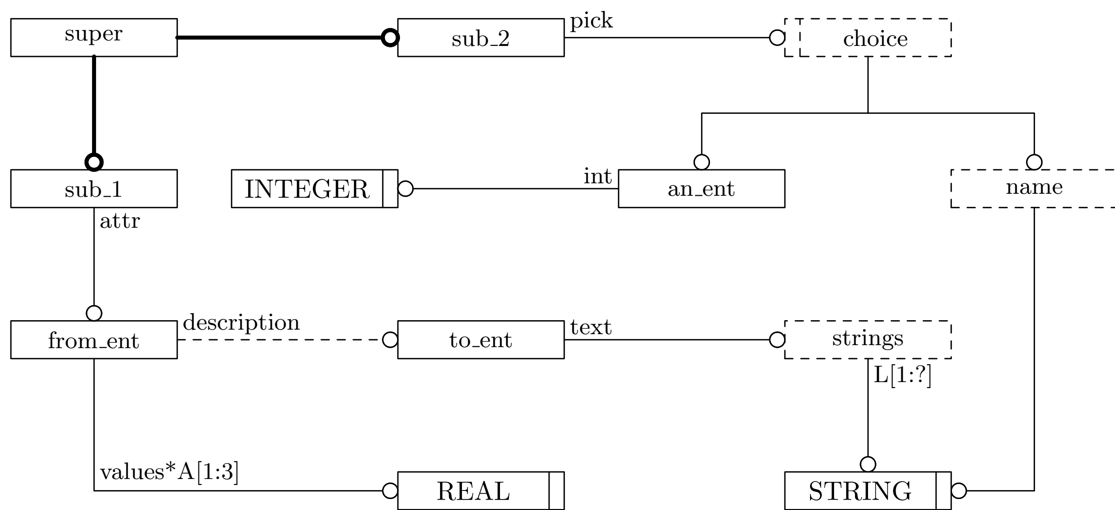


Figure D.17 – Complete entity level diagram of the example in J.2 on page 243 (Page 1 of 1)

D.5.2 Cardinalities

The attributes of entity and defined data types can be represented by aggregation data types (these are, 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 (that is, A, B, L, or S) is used, and the `OF` is omitted. Exceptionally, if an AGGREGATE data type is used, this is indicated on the relationship line for the attribute by using an empty pair of square brackets ([]) rather than a letter. 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 the example in J.2 on page 243 is fully displayed using EXPRESS-G in Figure D.17. 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;

— if an aggregation data type is constrained by a UNIQUE keyword, the first letter of the aggregation may be preceded by an asterisk.

 ©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

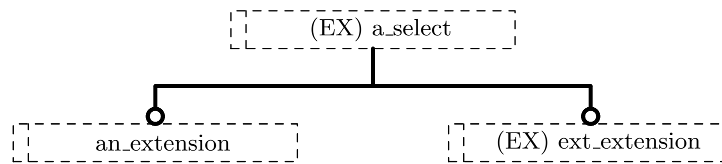


Figure D.18 – Extensible select data type diagram

D.5.4 Constructed and defined data types

A SELECT data type is represented by the select data type symbol (see Figure D.5) 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.5).

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.9) 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.17.

The extension relationship between an extensible constructed data type and a constructed data type based on it is denoted by a thick line linking the extensible data type to its extensions. Since there may be more than one extension to an extensible data type, the link linking the extensible data type to its extensions may branch. The extensible data type may have multiple lines leading to its extensions. The end of the line connected to the extensible data type has no end style. The end of the line connected to the extension is signified by an open circle.

NOTE 3 Two extensions to an extensible select data type, one of which is itself extensible, can be seen in figure D.18.

D.5.5 Entity data types

EXPRESS-G uses the solid box symbol (see Figure D.10) 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

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

attribute is distinguished from an explicit attribute by preceding the name of the attribute by the characters `DER` enclosed in parentheses, that is, `(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, that is, `(INV)`.

An inverse may not be the direct (simple) inverse of an explicit attribute. The entity referred to in the inverse attribute declaration may, among others, be a subtype of the entity that declared the direct relationship. Other indirect inverse relationships are described in 9.2.1.3. For such indirect inverse relationships the following rules shall apply:

— An inverse attribute is denoted by a normal line linking the entity in which the inverse attribute is defined, and the entity (or composition symbols for page or inter-schema references representing the entity) that represents the target of the inverse attribute. The target of an inverse attribute may be the entity declaring the explicit attribute, for which this is an inverse, or a subtype of that entity.

— The end of the line connected to the entity that contains the inverse attribute is signified by an open circle.

— The end of the line connected to the target entity has no end style.

— The name of the original explicit attribute and the entity that the attribute is declared in is placed adjacent to the line within parenthesis in the form "(entity_name.attribute_name)".

— The characters `INV` enclosed in parentheses, that is, `(INV)`, are placed before the name of the inverse attribute, which is also placed adjacent to the line.

— If the inverse attribute is constrained either by a where rule or a unique rule, the attribute name is preceded by a superscripted asterisk (*).

— If the attribute is defined by an aggregation data type, the aggregation data type is denoted as given in D.5.2, following the attribute name.

— If the inverse attribute is a redeclared attribute, it shall include the characters `RT` enclosed in parentheses, that is, `(RT)`, before the `(INV)` characters. If in this redeclaration the attribute is renamed, the new attribute name follows the original name, these being separated by the greater than symbol (>).

NOTE 1 Typical entity level diagrams are shown in Figure D.1 and Figure D.17.

NOTE 2 The indication of domain rules applied to attributes can be seen on the roles husband and maiden-name in Figure D.1.

NOTE 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

©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

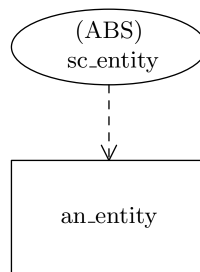


Figure D.19 – Symbol for denoting an ABSTRACT SUPERTYPE if the abstract constraint is defined within a SUBTYPE_CONSTRAINT

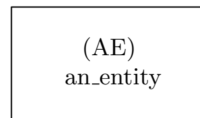


Figure D.20 – Symbol denoting an ABSTRACT ENTITY

ABSTRACT, the characters ABS, enclosed in parentheses, that is, (ABS), precede the name of the entity within the entity symbol box. If the ABSTRACT SUPERTYPE constraint is declared in a SUBTYPE_CONSTRAINT, the symbol (ABS) shall precede the name of the corresponding SUBTYPE_CONSTRAINT within the ellipse symbol for the SUBTYPE_CONSTRAINT, as shown in figure D.19.

When an entity is declared to be ABSTRACT (not an ABSTRACT SUPERTYPE, but an ABSTRACT ENTITY), the characters AE, enclosed in parentheses, that is, (AE), precede the name of the entity within the entity box symbol as shown in figure D.20.

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. If the ONEOF relation is defined by a SUBTYPE_CONSTRAINT, an asterisk shall precede the name of the SUBTYPE_CONSTRAINT in the ellipse symbol for the SUBTYPE_CONSTRAINT.

The AND relation may be indicated by a branching relationship line from the supertype to each of its subtypes that are in a AND relationship to each other, together with the character & being placed at the branching junction. If the AND relation is defined by a SUBTYPE_CONSTRAINT, an asterisk shall precede the name of the SUBTYPE_CONSTRAINT in the ellipse symbol for the SUBTYPE_CONSTRAINT.

The TOTAL_OVER constraint is presented in connection with the symbol for the SUBTYPE_CONSTRAINT as part of which it has been defined (see figure D.21). A dashed line shall be drawn between the ellipse of the SUBTYPE_CONSTRAINT and the constrained supertype entity box; this line style shall be used independent of the existence of a TOTAL_OVER constraint. Normal width solid lines shall be drawn between the ellipse of the SUBTYPE_CONSTRAINT and the entity boxes of the subtypes that provide the total coverage for the supertype. All these lines shall end with unclosed arrows at the entity boxes of the supertype and the subtypes that are included in the constraint. Instead of ending in entity boxes, the lines from a SUBTYPE_CONSTRAINT may end in composition symbols for page and inter-schema references.

©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

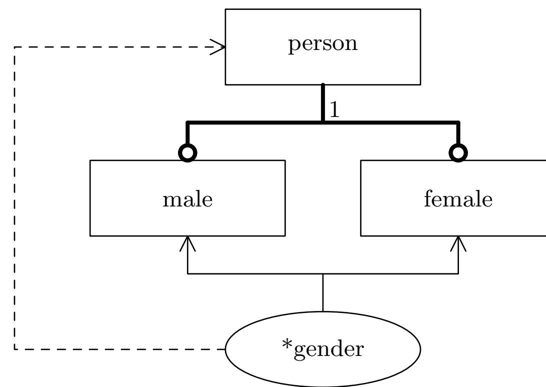


Figure D.21 – Example of the TOTAL_OVER coverage constraint

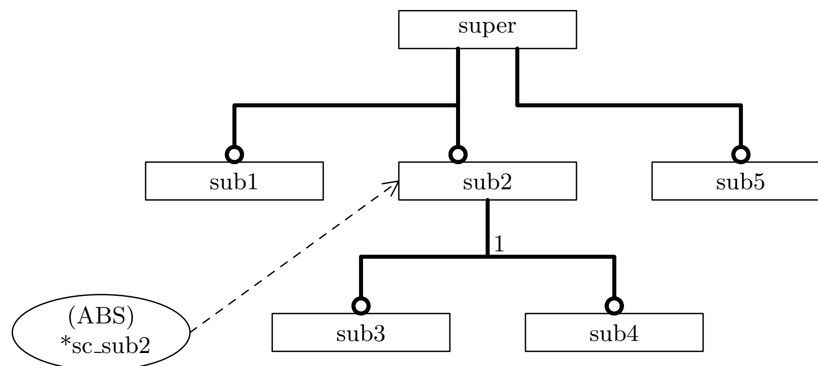


Figure D.22 – Complete entity level diagram of the inheritance graph from the example in J.3 on page 244 (Page 1 of 1)

EXAMPLE The following model is displayed in Figure D.21.

```
ENTITY person;
END_ENTITY;

ENTITY male SUBTYPE OF (person);
END_ENTITY;

ENTITY female SUBTYPE OF (person);
END_ENTITY;

SUBTYPE_CONSTRAINT gender FOR person;
  TOTAL_OVER(male, female);
  ONEOF(female, male);
END_SUBTYPE_CONSTRAINT;
```

NOTE 4 Figure D.22 provides an EXPRESS-G diagram of the example in J.3 on page 244, showing `sub2` as being an ABSTRACT SUPERTYPE.

NOTE 5 The diagram in Figure D.22 shows that the entities `sub1`, `sub2` and `sub5` are subtypes of the supertype `super`. 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.

EXPRESS permits the redeclaration of supertype attributes within a subtype, provided the

 ©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

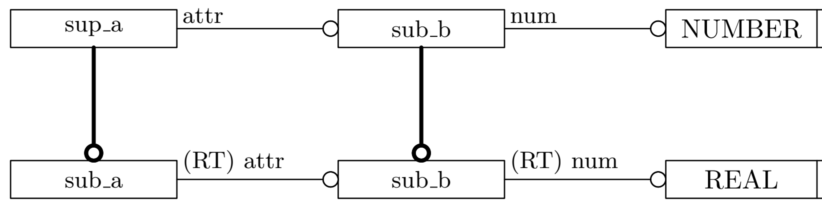


Figure D.23 – Complete entity level diagram of the example in J.4 on page 245 showing attribute redeclarations in subtypes (Page 1 of 1)

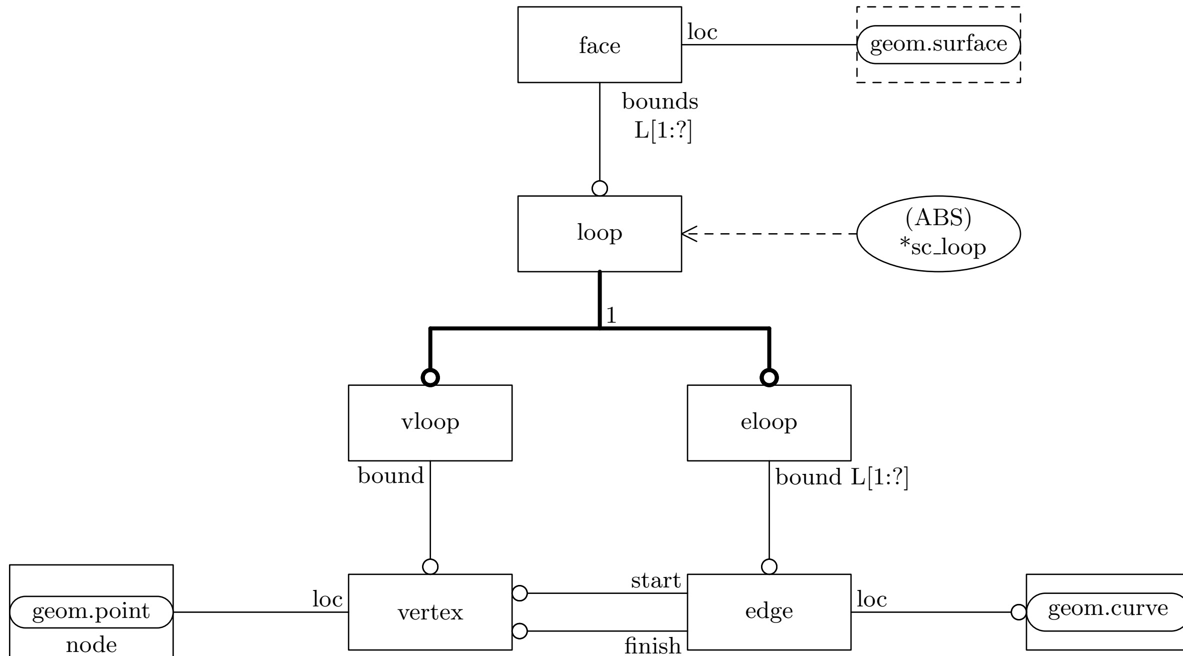


Figure D.24 – Complete entity level diagram of the `top` schema of example 1 on page 245 illustrating inter-schema references (Page 1 of 1)

redeclared attribute is a specialization of the supertype attribute type. If the attribute redeclaration also includes a rename of the attribute, the new name follows the original name, the two being separated by the greater than symbol (>). 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, that is, `(RT)`, before the name of the attribute.

NOTE 6 Figure D.23 illustrates some of the forms of attribute redeclaration, as given in the EXPRESS example in J.4 on page 245. 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, that 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.24 shows a entity level diagram of a single schema. The EXPRESS source for this diagram is given as example 1 on page 245. The complete diagram consists of two schemas, `top` and `geom` (see Figure D.25), 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

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

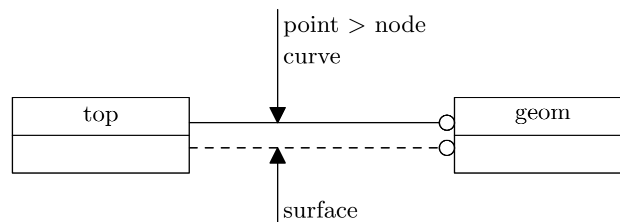


Figure D.25 – Complete schema level diagram of example 1 on page 245 (Page 1 of 1)

representation of the `top` schema in this example requires the inter-schema references shown.

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.25. The `top` schema has an interface to the `geom` schema. In particular, the `top` schema references the `surface` and uses the `curve` and `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, the page referencing symbols are used.

NOTE 2 Example 2 on page 246 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.26.

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.

 ©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

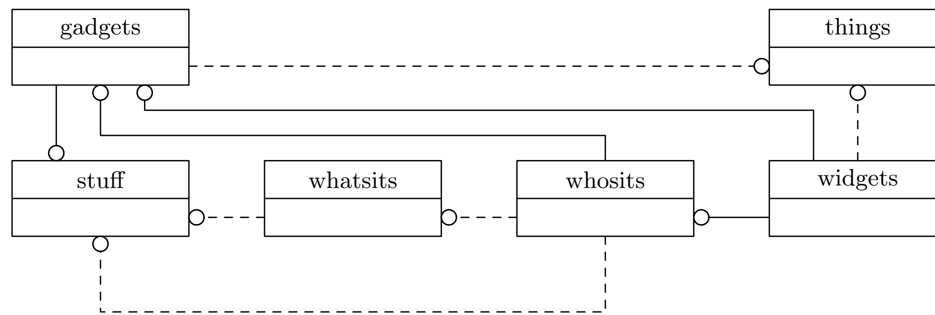


Figure D.26 – Complete schema level diagram of example 2 on page 246 (Page 1 of 1)

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 (that is, subtype and supertype) relationships are displayed;

h) All ABSTRACT SUPERTYPE constraints 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 (*);

p) All ABSTRACT ENTITY declarations are marked;

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

q) All TOTAL_OVER constraints are marked;

r) All relationships between extensible constructed data types and their extensions are displayed;

s) All the new and old names of attributes renamed during redeclaration are displayed;

t) All GENERIC_ENTITY SELECT constraints are displayed;

u) All GENERIC_ENTITY data types used within the schema are displayed.

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.

©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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.

©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

(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?:	............................:

©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Annex F
(normative)
Information object registration

F.1 Document identification

To provide for unambiguous identification of an information object in an open system, the object identifier

{ iso standard 10303 part(11) version(4) }

is assigned to this part of ISO 10303. The meaning of this value is defined in ISO/IEC 8824-1, and is described in ISO 10303-1.

F.2 Syntax identification

In order to provide for unambiguous identification of an information object in an open system, the object identifier

{ iso standard 10303 part(11) version(4) object(1) EXPRESS-syntax (1) }

is assigned to the syntax of EXPRESS. The meaning of this value is defined in ISO/IEC 8824-1, and is described in ISO 10303-1.

©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Annex G
(normative)
Generating a single schema from multiple schemas

G.1 Introduction

A data model specified via EXPRESS consists of at least one SCHEMA and may consist of more than one. There are many ways of composing a multi-schema specification, some of which are listed below.

— A portion of the specification may be defined in one SCHEMA which may be reused by another SCHEMA to be specialized or extended.

— A top level SCHEMA may import several SCHEMA to build a complete data specification.

— A complete specification may consist of several independent SCHEMA.

The original implementation methods included in ISO 10303 presupposed a data model consisting of a single SCHEMA, termed the longform schema. As long as the implementation methods included in ISO 10303 are not updated to this edition of ISO 10303-11 or implementations are not updated to work with the new implementation methods, it may be useful to convert a data specification that is in accordance to ISO 10303-11:2003 to a data specification that is in accordance to ISO 10303-11:1994. This annex specifies rules for how such a conversion may be done. The rules enable the conversion of an ISO 10303-11:2003 multi-schema specification to an ISO 10303-11:1994 longform schema. These rules are designed to result in a complete and consistent longform and to minimize the loss of semantics of the original data model.

G.2 Fundamental concepts

Entities that are in a supertype-subtype relationship form a directed graph which is constrained to be an acyclic (multi-rooted) tree. Similarly, SCHEMA that are linked via USE or REFERENCE interface specifications form a directed graph, which may be cyclic, where the nodes are SCHEMA and the edges are the interface specifications. In general, the specification of a data model consists of one or more such SCHEMA graphs. In particular, such a graph may have one root SCHEMA which is not the target of any interface specification, but from which all the other SCHEMA can be reached. The root schema can be considered to be representative of the graph. In other cases there can be one or more primary SCHEMA within the graph, where the other SCHEMA in the graph are only there to support the primary SCHEMA. The root and primary SCHEMA have a special role in the conversion process.

The input to the longform generation process is the set of root and primary SCHEMA of the graphs comprising the data model specification.

The output of the process is a single SCHEMA containing all the constructs in the input SCHEMA plus the necessary supporting constructs from the other SCHEMA in the set of graphs. The supporting constructs are those that are explicitly or implicitly interfaced into the root and primary SCHEMA.

The resulting longform is an almost semantically identical model without the USE and REFERENCE statements; referenced objects are brought directly into the schema. Information describing the originating SCHEMA is discarded. Objects in the interfaced schemas that are not directly referenced by the objects in the root or primary SCHEMA are discarded. Pruning and rewriting

 ©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

of some of the constructs of the input schemas takes place to omit the objects that are originally declared, but are not used in the final schema.

There are two stages to the conversion process, each of which involves semantic losses:

a) The multi-schema data specification is converted to an intermediate single schema specification. The following major transformations are applied:

   — Selectable items of a SELECT type are pruned to remove those items that are not interfaced into the schema. According to 11.4.2, selectable items are not implicitly interfaced as a result of the interfacing of the SELECT type itself. If the selectable items were left in the select list, and the objects were not visible in the schema, the compilation of the longform schema would result in errors.

   — SUBTYPE_CONSTRAINT's are pruned to reflect the pruning of the subtype/supertype graph as described in 11.4.3 and annex C.

   — RULE's are pruned to reflect the pruning of the subtype/supertype graph as described in 11.4.3 and annex C.

   — SCHEMA names in fully qualified attribute references are replaced with the SCHEMA name of the longform.

   — The knowledge of how the constructs were interfaced, that is, their visibility and instantiability, are transformed into rules. Information describing how the object was interfaced (the distinction between USE and REFERENCE), affects its instantiability and visibility; see clause 11 paragraphs 2 and 3.

   The following semantic information may be lost:

   — Loss of knowledge of the schema that each construct originates from.

   — Remarks that do not have remark tags (see 7.1.6.3) may be discarded.

   — The case of user identifiers may not be preserved.

b) The intermediate single schema representation is rewritten using only ISO 10303-11:1994 constructs to produce the final longform. The conversion from an ISO 10303-11:2003 intermediate representation shortform to an ISO 10303-11:1994 longform requires the removal or change of EXPRESS constructs that are not supported by ISO 10303-11:1994. In particular the following major actions are performed with the concomitant semantic losses.

   — EXTENSIBLE SELECT data types are resolved into SELECT data types according to ISO 10303-11:1994, which are not extensible.

   — EXTENSIBLE ENUMERATION data types are resolved into ENUMERATION data types according to ISO 10303-11:1994, which are not extensible.

   — SUBTYPE_CONSTRAINT statements are removed:

      — their SUPERTYPE constraints and ABSTRACT statements are converted into SUPERTYPE statements according to ISO 10303-11:1994 and are rewritten to remove types that would otherwise not appear in the longform schema;

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

— TOTAL_OVER constraints are replaced by RULE constructs.

— ABSTRACT ENTITY and GENERIC_ENTITY are converted into ABSTRACT SUPERTYPE constraints.

— RENAMED constructs are converted into DERIVE attributes.

— Errors shall be produced for empty SELECT data types.

The intermediate single schema specification is an artifact of the conversion process and shall have no validity outside the process.

These actions are described in more detail in the following sub-clauses.

G.3 Name munging

G.3.1 Name clashes

A SCHEMA defines a name scope in which the names of declarations are unique. Names of declarations in different SCHEMA are in different name scopes and, therefore, need not be unique across a collection of SCHEMA. The processes described below merge declarations from a set of SCHEMA into a single SCHEMA. In order to ensure name uniqueness within such a single SCHEMA, any names that are not unique across the original set of SCHEMA shall be modified to resolve the name clashes and ensure uniqueness in the merged SCHEMA.

Each non-unique name shall be preprended with the name of the SCHEMA in which it is declared, and the string '`_dot_`'. All occurrences of the name shall be modified.

EXAMPLE ENTITY `thing` is declared in SCHEMA `scha`, and TYPE `thing` is declared in SCHEMA `schb`. If these two schemas are to be merged, the ENTITY is renamed `scha_dot_thing` and the TYPE is renamed as `schb_dot_thing`.

G.3.2 Identifiers as strings

Within a SCHEMA, some strings may represent fully qualified names.

EXAMPLE 1 In the code fragment

```
IF 'THIS_SCHEMA.AN_ENTITY' IN TYPEOF(super)
```

the string `'THIS_SCHEMA.AN_ENTITY'` represents a fully qualified name.

When declarations that include such strings are moved from their original schema to another schema, the schema portion of such strings shall be modified to represent the schema into which they have been moved.

EXAMPLE 2 A declaration called `whatsit` in SCHEMA `scha` is copied into another SCHEMA called `schb`. Another declaration which includes the string `'SCHA.WHATSIT'` is also copied into `schb`. In SCHEMA `schb` the string will appear as `'SCHB.WHATSIT'`.

 ©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

G.4 Stage 1: multi-schema to intermediate schema conversion

G.4.1 Introduction

This is the first stage in generating an ISO 10303-11:1994 single schema data model specification from a multi-schema data model specification. The result of this stage is a data model specification in the form of an ISO 10303-11:2003 single schema. For pedagogical purposes call this the `artifact` SCHEMA.

The input to the stage 1 process is the root and primary schemas for the data model specification. The initial data model specification shall be referentially complete; that is, there are no unidentified references.

The method of identifying the input schemas is out of scope, and left to an implementor.

G.4.2 Primary population

Create a new intermediate SCHEMA called `artifact`, and copy into it all the declarations and tagged remarks in the root and primary schemas. The `artifact` schema shall not have a `schema_version_id`. Any name clashes between the declarations and tagged remarks in `artifact` shall be resolved, and any string identifier representations modified appropriately.

Remove any duplicates in USE and REFERENCE specifications. If any item appears in both a USE and a REFERENCE specification, delete the item from the REFERENCE declaration — a USE interface specification takes precedence over a REFERENCE specification (see 11.3).

The effect of a USE specification is to enable items in foreign schema be treated as though they were declared locally (see 11.1). Copy all such items and the tagged remarks that relate to them into the `artifact` schema, resolving any name clashes and modifying string identifier representations.

When an interfaced item is subject to a renaming (see 11.1 and 11.2), it shall be copied under its original name, and identifier references shall be changed accordingly.

Then delete all USE specifications from the `artifact` schema as all items that have been identified via the USE specifications have been copied into the schema.

NOTE The `artifact` schema will now contain all those items that were originally declared in the input root and primary schemas, plus the items that were USE interfaced into the input schemas, plus the (modified) REFERENCE specifications from the input schemas. There will be no USE specifications.

EXAMPLE 1 This example illustrates copying of renamed items. Given:

```
SCHEMA sch;
  USE FROM second (alfred AS alf);
  REFERENCE FROM second (bert AS herbert);

  ENTITY joe;
    attr1 : alf;
    attr2 : herbert;
  END_ENTITY;
...
```

and

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

```
SCHEMA short;
  USE sch;
  ...
END_SCHEMA;
```

then if `short` was a root schema input to the algorithm and assuming `alfred` and `bert` are ENTITY declarations, the `artifact` schema will contain

```
SCHEMA artifact;

  ENTITY joe;
    attr1 : alfred;
    attr2 : bert;
  END_ENTITY;

  ENTITY alfred ...
  ENTITY bert ...
...
```

EXAMPLE 2 This example illustrates name munging to resolve name clashes. The original specification contains three schemas:

```
SCHEMA s1;

  ENTITY creature;
    -- attributes
  END_ENTITY;

  -- other declarations
END_SCHEMA; -- end s1

SCHEMA farming;
  USE FROM s1 (creature);

  ENTITY dog SUBTYPE OF creature;
    -- attributes
  END_ENTITY;

  ENTITY shepherd;
    dogs : SET OF dog;
  END_ENTITY;

  -- other declarations
END_SCHEMA; -- end farming

SCHEMA pet_shows;
  USE FROM s1 (creature);

  ENTITY pet SUBTYPE OF (creature);
    -- attributes
  END_ENTITY;

  ENTITY dog SUBTYPE OF pet;
    -- attributes
  END_ENTITY;

  ENTITY dog_show;
    dogs : SET [1:?] OF dog;
    -- other attributes
```

 ©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
  END_ENTITY;

  -- other declarations
END_SCHEMA; -- end pet_shows
```

Taking `farming` and `pet_shows` as the primary schemas, the resulting intermediate schema is:

```
SCHEMA artifact;

  ENTITY creature;
    -- attributes
  END_ENTITY;

  ENTITY farming_dot_dog SUBTYPE OF creature;
    -- attributes
  END_ENTITY;

  ENTITY shepherd;
    dogs : SET OF farming_dot_dog;
  END_ENTITY;

  ENTITY pet SUBTYPE OF (creature);
    -- attributes
  END_ENTITY;

  ENTITY pet_shows_dot_dog SUBTYPE OF pet;
    -- attributes
  END_ENTITY;

  ENTITY dog_show;
    dogs : SET [1:?] OF pet_shows_dot_dog;
    -- other attributes
  END_ENTITY;

  -- other declarations
END_SCHEMA; -- end artifact
```

G.4.3 Secondary population

Examine each item that has been identified in the REFERENCE specifications in the `artifact` schema. If it is required for referential completeness, copy the declaration and the tagged remarks that relate to this declaration from their original schema into the `artifact` schema. If the item is an entity data type, maintain the semantics of the REFERENCE statement that the item shall only be instantiated if it is referenced by another item by applying the following procedure:

— for the first such item create in the `artifact` schema the following rule and function:

```
RULE validate_dependently_instantiable_entity_data_types FOR
    (<list this first and all subsequent relevant referenced
      entity data types here>);
LOCAL
  number_of_input_instances : INTEGER;
  previous_in_chain         : LIST OF GENERIC := [];
  set_of_input_types        : SET OF STRING := [];
  all_instances             : SET OF GENERIC := [];
END_LOCAL;
```


Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

```
all_instances := <make a union of all implicit populations of the
                  FOR-clause>;
number_of_input_instances := SIZEOF(all_instances);
(* Collect all type strings of all FOR instances into one set. *)
REPEAT i:=1 TO number_of_input_instances;
  set_of_input_types := set_of_input_types + TYPEOF(all_instances[i]);
END_REPEAT;

WHERE
 WR1: dependently_instantiated(all_instances, set_of_input_types,
                               previous_in_chain);
END_RULE;

FUNCTION dependently_instantiated(
        set_of_input_instances : SET OF GENERIC:igen;
        set_of_input_types     : SET OF STRING;
        previous_in_chain      : LIST OF GENERIC:cgen): BOOLEAN;
(*"dependently_instantiated" To test whether all instances in the
  input set_of_input_instances are referenced by independently
  instantiable instances. If so, this function returns true.
  Set_of_input_types includes the type strings for all input instances.
  The instances in previous_in_chain are used to detect cyclic
  references during recursive calls to this function. The parameter
  lists already tested instances in a chain of references.
 *)
LOCAL
  number_of_input_instances     : INTEGER;
  number_of_referring_instances : INTEGER;
  bag_of_referring_instances    : BAG OF GENERIC:igen := [];
  dependently_instantiated_flag : BOOLEAN;
  previous_in_chain_plus        : LIST OF GENERIC:cgen := [];
  result                        : BOOLEAN := true;
  set_of_types                  : SET OF STRING := [];
END_LOCAL;

IF EXISTS(set_of_input_instances) THEN
  number_of_input_instances := SIZEOF(set_of_input_instances);
  (* Add the declared type of bag_of_referring_instances to the set of
     types of the REFERENCEd instances for the subset comparison later.
   *)
  set_of_input_types := set_of_input_types + 'GENERIC';
  REPEAT i:=1 TO number_of_input_instances;
    (* Determine all references to the current input instance. *)
    bag_of_referring_instances := USEDIN (set_of_input_instances[i] , '');
    IF EXISTS(bag_of_referring_instances) THEN
      number_of_referring_instances := SIZEOF(bag_of_referring_instances);
      dependently_instantiated_flag := false;
      REPEAT j:=1 TO number_of_referring_instances;
        (* Determine the type strings of the current referencing instance.
         *)
        set_of_types := TYPEOF(bag_of_referring_instances[j]);
```

 ©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

```
        (* If the referencing instance is of one of the types of the
           only dependently instantiable select items, the current input
           instance may still be invalidly instantiated.
           Otherwise it is OK, and the next input instance is tested.
         *)
        IF set_of_types <= set_of_input_types THEN -- subset operator
          (* The referring instance is of one of the restricted types.
             However, it may itself be referred to by a valid instance;
             then also the current instance would be valid.
             Thus, call this function recursively with the referring
             instance as input.
             To avoid an infinite loop in case a set of instances
             reference each other in a closed loop, test first whether
             the current referencing instance is in the list of
             previously processed chain members.
           *)
          IF NOT (bag_of_referring_instances[j] IN previous_in_chain) THEN
            previous_in_chain_plus := previous_in_chain +
                                      set_of_input_instances[i];
            IF dependently_instantiated([bag_of_referring_instances[j]],
                                        set_of_input_types,
                                        previous_in_chain_plus) THEN
              dependently_instantiated_flag := true;
              ESCAPE; -- dependently instantiated; next input instance
            ELSE
              (* Not dependently instantiated: go to next referring
                 instance. *)
              SKIP;
            END_IF;
          END_IF;
        ELSE
          dependently_instantiated_flag := true;
          ESCAPE; -- dependently instantiated; take next input instance
        END_IF;
      END_REPEAT;
      IF NOT dependently_instantiated_flag THEN
        RETURN(false);
      END_IF;
    ELSE
      RETURN(false); -- not referenced at all => invalidly instantiated
    END_IF;
  END_REPEAT;
ELSE
  RETURN(false); -- no input
END_IF;

RETURN(true);
END_FUNCTION; -- end dependently_instantiated
```

— add to the FOR-clause and to the right side of the assignment statement with `all_instances` on the left side (both locations are marked by angular brackets `< >`) the names of the first

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

and all following such entity data types;

— add in these two locations also such entity data types that the already identified ones require for their referential completeness and that are not included in the longform elsewhere as independently instantiable entity data types;

— if one or several of the primary schemas are longforms that were created based on this procedure and, thus, already contain a rule `validate_dependently_instantiable_entity_data_types`, add the relevant contents of their FOR-clauses to the one created here. Relevant contents are entity data types that remain dependently instantiable also after the current longform generation and that are not included in the new rule already.

If the item is not required for referential completeness, it shall not be copied. Name clashes shall be resolved and string identifiers modified.

NOTE 1 Not copying declarations that are not required for completeness maintains the REFERENCE semantics of those declarations.

EXAMPLE 1 In this example the original model consists of two schemas, one of which has no interface specifications whereas the other one, the dedicated longform schema, has a REFERENCE specification:

```
SCHEMA export;

  ENTITY a;
    a1: STRING;
  END_ENTITY;

  ENTITY b;
    b1: STRING;
  END_ENTITY;

END_SCHEMA; -- end export

SCHEMA import;

  REFERENCE FROM export (a, b); -- only dependently instantiable!

  ENTITY ref;
    aref: a; -- instantiation is dependent on entity data type ref
    bref: b; -- instantiation is dependent on entity data type ref
  END_ENTITY;

END_SCHEMA; -- end import
```

Using `import` as the root schema, the intermediate schema is:

```
SCHEMA artifact;

  ENTITY a;
    a1: STRING;
  END_ENTITY;

  ENTITY b;
    b1: STRING;
  END_ENTITY;
```

 ©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
ENTITY ref;
  aref: a; -- instantiated dependent on entity data type ref
  bref: b; -- instantiated dependent on entity data type ref
END_ENTITY;

RULE validate_dependently_instantiable_entity_data_types FOR
    (a, b); -- !! here a and b have been added !!
LOCAL
  number_of_input_instances : INTEGER;
  previous_in_chain         : LIST OF GENERIC := [];
  set_of_input_types        : SET OF STRING := [];
  all_instances             : SET OF GENERIC := [];
END_LOCAL;

all_instances := a+b; -- !! here a and b have been added !!
number_of_input_instances := SIZEOF(all_instances);
(* Collect all type strings of all FOR instances into one set. *)
REPEAT i:=1 TO number_of_input_instances;
  set_of_input_types := set_of_input_types + TYPEOF(all_instances[i]);
END_REPEAT;

WHERE
 WR1: dependently_instantiated(all_instances, set_of_input_types,
                               previous_in_chain);
END_RULE;

FUNCTION dependently_instantiated(
        set_of_input_instances : SET OF GENERIC:igen;
        set_of_input_types     : SET OF STRING;
        previous_in_chain      : LIST OF GENERIC:cgen): BOOLEAN;
(*"dependently_instantiated" To test whether all instances in the
  input set_of_input_instances are referenced by independently
  instantiable instances. If so, this function returns true.
  Set_of_input_types includes the type strings for all input instances.
  The instances in previous_in_chain are used to detect cyclic
  references during recursive calls to this function. The parameter
  lists already tested instances in a chain of references.
 *)
LOCAL
  number_of_input_instances     : INTEGER;
  number_of_referring_instances : INTEGER;
  bag_of_referring_instances    : BAG OF GENERIC:igen := [];
  dependently_instantiated_flag : BOOLEAN;
  previous_in_chain_plus        : LIST OF GENERIC:cgen := [];
  result                        : BOOLEAN := true;
  set_of_types                  : SET OF STRING := [];
END_LOCAL;

IF EXISTS(set_of_input_instances) THEN
  number_of_input_instances := SIZEOF(set_of_input_instances);
  (* Add the declared type of bag_of_referring_instances to the set of
     types of the REFERENCEd instances for the subset comparison later.
   *)
  set_of_input_types := set_of_input_types + 'GENERIC';
  REPEAT i:=1 TO number_of_input_instances;
    (* Determine all references to the current input instance. *)
    bag_of_referring_instances := USEDIN (set_of_input_instances[i] , '');
    IF EXISTS(bag_of_referring_instances) THEN
      number_of_referring_instances := SIZEOF(bag_of_referring_instances);
```


Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

```
        dependently_instantiated_flag := false;
        REPEAT j:=1 TO number_of_referring_instances;
          (* Determine the type strings of the current referencing instance.
           *)
          set_of_types := TYPEOF(bag_of_referring_instances[j]);
          (* If the referencing instance is of one of the types of the
             only dependently instantiable select items, the current input
             instance may still be invalidly instantiated.
             Otherwise it is OK, and the next input instance is tested.
           *)
          IF set_of_types <= set_of_input_types THEN -- subset operator
            (* The referring instance is of one of the restricted types.
               However, it may itself be referred to by a valid instance;
               then also the current instance would be valid.
               Thus, call this function recursively with the referring
               instance as input.
               To avoid an infinite loop in case a set of instances
               reference each other in a closed loop, test first whether
               the current referencing instance is in the list of
               previously processed chain members.
             *)
            IF NOT (bag_of_referring_instances[j] IN previous_in_chain) THEN
              previous_in_chain_plus := previous_in_chain +
                                        set_of_input_instances[i];
              IF dependently_instantiated([bag_of_referring_instances[j]],
                                          set_of_input_types,
                                          previous_in_chain_plus) THEN
                dependently_instantiated_flag := true;
                ESCAPE; -- dependently instantiated; next input instance
              ELSE
                (* Not dependently instantiated: go to next referring
                   instance. *)
                SKIP;
              END_IF;
            END_IF;
          ELSE
            dependently_instantiated_flag := true;
            ESCAPE; -- dependently instantiated; take next input instance
          END_IF;
        END_REPEAT;
        IF NOT dependently_instantiated_flag THEN
          RETURN(false);
        END_IF;
      ELSE
        RETURN(false); -- not referenced at all => invalidly instantiated
      END_IF;
    END_REPEAT;
  ELSE
    RETURN(false); -- no input
  END_IF;

  RETURN(true);
  END_FUNCTION; -- end dependently_instantiated

END_SCHEMA; -- end artifact
```

When a foreign declaration is interfaced into a schema, other declarations may be implicitly interfaced (see 11.4). Copy those implicitly interfaced declarations that are required for referential completeness of the `artifact` schema including their tagged remarks from their original

 ©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

schema into `artifact`, resolving name clashes and modifying string identifier representations.

NOTE 2 Copying implicitly interfaced items may be a recursive process.

EXAMPLE 2 In this example the original model consists of two schemas:

```
SCHEMA export;

  TYPE colour = EXTENSIBLE ENUMERATION;
  END_TYPE;

  TYPE stop_light = ENUMERATION BASED_ON colour WITH (red, yellow, green);
  END_TYPE;

END_SCHEMA; -- end export

SCHEMA import;

  USE FROM export (stop_light);
  REFERENCE FROM export (colour);

  TYPE canadian_flag = ENUMERATION BASED_ON colour WITH (red, white);
  END_TYPE;

  -- other declarations dependent on canadian_flag and stop_light
  ...

END_SCHEMA; -- end import
```

Using `import` as the root schema, the intermediate schema is:

```
SCHEMA artifact;

  TYPE colour = EXTENSIBLE ENUMERATION;
  END_TYPE;

  TYPE stop_light = ENUMERATION BASED_ON colour WITH (red, yellow, green);
  END_TYPE;

  TYPE canadian_flag = ENUMERATION BASED_ON colour WITH (red, white);
  END_TYPE;

  -- other declarations dependent on canadian_flag and stop_light
  ...

END_SCHEMA; -- end artifact
```

EXAMPLE 3 In this example the original model consists of four schemas:

```
SCHEMA s1;

  TYPE general_approval = EXTENSIBLE ENUMERATION OF (approved, rejected);
  END_TYPE;

END_SCHEMA; -- end s1

SCHEMA s2;
```


Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

```
  USE FROM s1 (general_approval);

  TYPE domain2_approval = EXTENSIBLE ENUMERATION BASED_ON general_approval WITH
       (pending);
  END_TYPE;

END_SCHEMA; -- end s2

SCHEMA s3;

  USE FROM s1 (general_approval);

  TYPE domain3_approval = EXTENSIBLE ENUMERATION BASED_ON general_approval WITH
       (cancelled);
  END_TYPE;

END_SCHEMA; -- end s3

SCHEMA s4;

  USE FROM s2 (domain2_approval);
  REFERENCE FROM s3 (domain3_approval);

  TYPE specific_approval = ENUMERATION BASED_ON domain2_approval WITH
       (rework);
  END_TYPE;

  -- other declarations dependent on these types
  ...

END_SCHEMA; -- end s4
```

Using s4 as the root schema, the intermediate schema is:

```
SCHEMA artifact;

  TYPE specific_approval = ENUMERATION BASED_ON domain2_approval WITH
       (rework);
  END_TYPE;

  TYPE domain3_approval = EXTENSIBLE ENUMERATION BASED_ON general_approval WITH
       (cancelled);
  END_TYPE;

  TYPE domain2_approval = EXTENSIBLE ENUMERATION BASED_ON general_approval WITH
       (pending);
  END_TYPE;

  TYPE general_approval = EXTENSIBLE ENUMERATION OF (approved, rejected);
  END_TYPE;

  -- other declarations dependent on these types
  ...

END_SCHEMA; -- end artifact
```

G.4.4 Prune

There are limits on the implicitly interfaced declarations (see 11.4).

 ©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

EXAMPLE 1 A SUPERTYPE ENTITY does not implicitly interface any SUBTYPE of that entity.

EXAMPLE 2 A SELECT type does not implicitly interface any of its selectable items.

EXAMPLE 3 A RULE does not implicitly interface any of the ENTITY types specified in its parameter list.

Due to the limitations on implicit interfaces, there may be declarations in the `artifact` schema that are not referentially complete.

Supertype constraint expressions shall be pruned according to annex C.

A RULE where not all its parameters are visible shall be pruned by deleting it and the tagged remarks that relate to it

A RULE may call a FUNCTION or PROCEDURE. Any FUNCTION or PROCEDURE within the `artifact` schema that is not called by a declaration in the schema shall be deleted including its tagged remarks.

A SELECT type shall be pruned by deleting from the select list each selectable item that is not visible to the `artifact` schema.

NOTE 1 Pruning a SELECT type may result in an empty select list.

Delete any REFERENCE specifications.

NOTE 2 At this point the `artifact` schema should be referentially complete. All declarations, except for those in the original input schema or USE interfaced by those schema, should be necessary for the referential completeness of those original declarations.

EXAMPLE 4 This example illustrates pruning a SELECT type. The original model consists of two schemas;

```
SCHEMA export;

  TYPE attachment_method = EXTENSIBLE SELECT(nail, screw);
  END_TYPE;

  ENTITY nail;
  END_ENTITY;

  ENTITY screw;
  END_ENTITY;

END_SCHEMA; -- end export

SCHEMA import;

  USE FROM export (attachment_method,
                   nail);

  TYPE permanent_attachment = SELECT BASED_ON attachment_method WITH
       (glue, weld);
  END_TYPE;

  TYPE simple_attachment = SELECT BASED_ON attachment_method WITH
```


Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

```
      (needle, tape);
  END_TYPE;

  -- declarations of glue and others
  ...

END_SCHEMA; -- end import
```

The import schema is used as the root schema. As screw is neither explicitly nor implicitly interfaced into import, it does not appear in the intermediate schema, which is:

```
SCHEMA artifact;

  TYPE attachment_method = EXTENSIBLE SELECT(nail);
  END_TYPE;

  ENTITY nail;
  END_ENTITY;

  TYPE permanent_attachment = SELECT BASED_ON attachment_method WITH
      (glue, weld);
  END_TYPE;

  TYPE simple_attachment = SELECT BASED_ON attachment_method WITH
      (needle, tape);
  END_TYPE;

  -- declarations of glue and others
  ...

END_SCHEMA; -- end artifact
```

Notice that the original attachment_method SELECT in the export schema has been pruned by removing screw from the selectable items.

EXAMPLE 5 This example is an illustration of where pruning a SELECT type results in an empty list of selectable items. The original specification has three schemas:

```
SCHEMA s1;

  TYPE t11 = EXTENSIBLE SELECT
      (t12, t13);
  END_TYPE;

  -- declarations of t12, t13 and others
  ...

END_SCHEMA; -- end s1

SCHEMA s2;
  REFERENCE FROM s1 (t11);

  ENTITY e21;
    attr : t11;
  END_ENTITY;

  -- other declarations
  ...
```

 ©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

```
END_SCHEMA; -- end s2

SCHEMA s3;
  USE FROM s1 (t11);
  USE FROM s2 (e21);

  TYPE t31 = SELECT BASED_ON t11 WITH
        (t32, t33);
  END_TYPE;

  -- declarations of t32, t33 and others
  ...

END_SCHEMA; -- end s3
```

Using `s2` as the root schema, the resulting intermediate schema is:

```
SCHEMA artifact;

  ENTITY e21;
    attr : t11;
  END_ENTITY;

  TYPE t11 = EXTENSIBLE SELECT;
  END_TYPE;

  -- other declarations
  ...
END_SCHEMA; -- end artifact
```

Schema `s2` interfaces `t11` from schema `s1`, but does not interface any of the items in its selectable list. After pruning, the list is empty and disappears from the final representation of the intermediate schema.

EXAMPLE 6 The original model consists of two schemas. The example illustrates pruning a TOTAL_OVER constraint and a RULE.

```
SCHEMA s1;

  ENTITY e1;
  END_ENTITY;

  ENTITY e2 SUBTYPE OF (e1);
  END_ENTITY;

  ENTITY e3 SUBTYPE OF (e1);
  END_ENTITY;

  SUBTYPE_CONSTRAINT sc_total_over FOR e1;
    TOTAL_OVER (e2, e3);
  END_SUBTYPE_CONSTRAINT;

  RULE e2_and_e3 (e2, e3);
    -- rule body
  END_RULE;

END_SCHEMA; -- end s1

SCHEMA import;
  USE FROM s1 (e1, e2);
```


Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

```
END_SCHEMA; -- end import
```

Using `import` as the root schema, the intermediate schema before any pruning is:

```
SCHEMA artifact;

  ENTITY e1;
  END_ENTITY;

  ENTITY e2 SUBTYPE OF (e1);
  END_ENTITY;

  SUBTYPE_CONSTRAINT sc_total_over FOR e1;
    TOTAL_OVER (e2, e3);
  END_SUBTYPE_CONSTRAINT;

  RULE e2_and_e3 (e2, e3);
    -- rule body
  END_RULE;

END_SCHEMA; -- end artifact
```

Notice that entity `e3` does not appear.

After pruning, the final form of the intermediate schema reduces to:

```
SCHEMA artifact;

  ENTITY e1;
  END_ENTITY;

  ENTITY e2 SUBTYPE OF (e1);
  END_ENTITY;

  SUBTYPE_CONSTRAINT sc_total_over FOR e1;
    TOTAL_OVER (e2);
  END_SUBTYPE_CONSTRAINT;

END_SCHEMA; -- end artifact
```

As entity `e3` does not appear in the `artifact` schema, the SUBTYPE_CONSTRAINT is pruned accordingly. Similarly, the RULE is deleted.

EXAMPLE 7 This example is based on `SCHEMA example` (see example 1 on page 171) in annex B and illustrates pruning supertype expressions. The original schemas according to ISO 10303-11:2003 are:

```
SCHEMA example;

  ENTITY p;
  END_ENTITY;

  SUBTYPE_CONSTRAINT p_subs FOR p;
    ONEOF(m, f) AND ONEOF(c, a);
  END_SUBTYPE_CONSTRAINT;

  ENTITY m SUBTYPE OF (p);
  END_ENTITY;
```

 ©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
  ENTITY f SUBTYPE OF (p);
  END_ENTITY;

  ENTITY c SUBTYPE OF (p);
  END_ENTITY;

  ENTITY a ABSTRACT SUBTYPE OF (p);
  END_ENTITY;

  SUBTYPE_CONSTRAINT no_li FOR a;
    ONEOF(l, i);
  END_SUBTYPE_CONSTRAINT;

  ENTITY l SUBTYPE OF (a);
  END_ENTITY;

  ENTITY i SUBTYPE OF (a);
  END_ENTITY;

END_SCHEMA; -- end example

SCHEMA import;
  USE FROM example(l);
  REFERENCE FROM example(m);
END_SCHEMA; -- end import
```

Using `import` as the root schema, the intermediate schema before pruning is:

```
SCHEMA artifact;

  ENTITY p;
  END_ENTITY;

  SUBTYPE_CONSTRAINT p_subs FOR p;
    ONEOF(m, f) AND ONEOF(c, a);
  END_SUBTYPE_CONSTRAINT;

  ENTITY m SUBTYPE OF (p);
  END_ENTITY;

  ENTITY a ABSTRACT SUBTYPE OF (p);
  END_ENTITY;

  SUBTYPE_CONSTRAINT no_li FOR a;
    ONEOF(l, i);
  END_SUBTYPE_CONSTRAINT;

  ENTITY l SUBTYPE OF (a);
  END_ENTITY;

END_SCHEMA; -- end artifact
```

The entities `f`, `c`, and `i` are not in the schema, so the supertype expressions in `p_subs` and `no_li` have to be reduced according to annex C. Taking the expression in `p_subs` first, that is:
`ONEOF(m, f) AND ONEOF(c, a);`
The first reduction (`ONOEOF(A, <>) => ONEOF(A)`) results in the reduced expression:
`ONEOF(m) AND ONEOF(a);`
This can be reduced once more (via `ONEOF(A) => A`) to produce the final reduced expression:
`m AND a;`

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Thus, after pruning, p_subs is

```
  SUBTYPE_CONSTRAINT p_subs FOR p;
    m AND a;
  END_SUBTYPE_CONSTRAINT;
```

The expression in no_li is:
ONEOF(l, i);
and applying the reductions the final result for no_li is:

```
  SUBTYPE_CONSTRAINT no_li FOR a;
    l;
  END_SUBTYPE_CONSTRAINT;
```

This is a vacuous constraint, and so the SUBTYPE_CONSTRAINT as a whole and potentially related tagged remarks can be eliminated.

The final result after pruning is:

```
SCHEMA artifact;

  ENTITY p;
  END_ENTITY;

  SUBTYPE_CONSTRAINT p_subs FOR p;
    m AND a;
  END_SUBTYPE_CONSTRAINT;

  ENTITY m SUBTYPE OF (p);
  END_ENTITY;

  ENTITY a ABSTRACT SUBTYPE OF (p);
  END_ENTITY;

  ENTITY l SUBTYPE OF (a);
  END_ENTITY;

END_SCHEMA; -- end artifact
```

G.4.5 Schema names and versions

The names of the original schemas from which items have been copied into the intermediate schema may be kept as an embedded remark. If the name of any schema is kept, the names of all the schemas shall be kept.

The embedded remark shall have the following format (where \n indicates the end of a line, the text between <...> is variable, and [...] indicates an optional element):

```
(* Original 2003 schemas: \n
   schema = <schema_id> [schema_version_id = '<version>'] ; \n
   ...
*)
```

If a schema that includes a schema_version_id is listed in the embedded remark, the schema_version_id shall be listed as well.

 ©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

The order in which the schemas are listed, has no significance.

EXAMPLE 1 When one of the original schemas is:
`SCHEMA geometry_schema;`
an embedded remark recording this would include:
`schema = geometry_schema;`

EXAMPLE 2 The embedded remark where there are two original schemas, as:

```
SCHEMA schema_one;
...
SCHEMA schema_two 'version 4';
...
```

would be:

```
(* Original 2003 schemas:
   schema = schema_two schema_version_id = 'version 4';
   schema = schema_one;
*)
```

G.5 Stage 2: convert intermediate schema to ISO 10303-11:1994

G.5.1 Introduction

This is the second stage in generating an ISO 10303-11:1994 single schema data model specification from a multi-schema data model representation.

The input to this stage is a referentially complete single schema data model specification that has been generated according to G.4. The output is a single referentially complete schema that has no constructs that are not specified in ISO 10303-11:1994.

The following rules specify how the constructs in the input schema shall be reduced to those defined in ISO 10303-11:1994.

For pedagogical purposes call the input schema `artifact` and the output SCHEMA `longform`.

G.5.2 Initialisation

Create a new SCHEMA called `longform`. Copy all the declarations and remarks from the intermediate schema into the `longform` schema. Any string identifier representations shall be modified appropriately.

G.5.3 Conversion of extensible constructed data types

Extensible select data types and extensible enumeration data types (see 8.4) are not part of ISO 10303-11:1994. A tree of either of these types has to be reduced to a single non-extensible type. Constructed data types that are not part of such a tree shall simply be copied from the `artifact` schema to the `longform` schema.

To reduce a tree of extensible types perform the following conversions:

— each extensible or extending data type shall be replaced by a data type of the same name that is neither extensible nor an extension of any data type;

©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

— the list items specified in each constructed data type shall be the set of items within the domain of the data type as evaluated with respect to the `longform` schema;

— to maintain the dependencies among the extensible and extending data types their target constructs shall be related; depending data types, these are the extending ones, shall be created as defined data types that use the extensible data type as underlying type; the elements of the underlying data type that are not valid in the defined data type shall be constrained using local rules;

— in the case that a defined data type in the schema that is visible in the context schema has an extensible or extending data type as its underlying data type, add a WHERE clause to that defined data type; this shall eliminate all items that result from the conversion and that are not valid list items in the set of schemas that are visible in the context schema; see below for examples;

— this procedure may copy constructed data types into the target that are not referenced by any other data type in the longform. Such data types and their tagged remarks shall be eliminated from the longform.

The following sub-clauses give details on how to implement these rules for EXTENSIBLE ENUMERATION and EXTENSIBLE SELECT data types.

G.5.3.1 Extensible enumeration

EXTENSIBLE ENUMERATION (see 8.4.1) shall be converted according to the procedure above to enumeration data types that are not extensible. The names of the constructs shall be maintained.

If all of the enumeration items of an extending enumeration data type are valid in the context of the target longform, the extending enumeration data type shall be converted to a defined data type that has as underlying data type the target enumeration data type of the extensible enumeration that it is based on. The name of the defined data type in the target model shall be the name of the corresponding enumeration data type in the source.

If the target defined data type shall exclude enumeration items that are invalid in its context, but that are specified for its underlying enumeration, proceed as follows:

— create an intermediate defined data type and give it a name that is unique in the target longform;

— use for its underlying data type the underlying data type of the extending enumeration data type;

— create another defined data type and give it the name of the extending enumeration data type that shall be converted;

— use the intermediate defined data type as its underlying type;

— exclude enumeration items that are invalid in the target context, but that are specified for the enumeration under conversion, from instantiation in the target schema by local rules (see example 3 below);

— one local rule shall be created for each enumeration item that shall be excluded.

 ©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

NOTE See rule (j) in 8.4.1 concerning the use of local rules in type declarations that declare enumeration data types.

EXAMPLE 1 The intermediate schema generated according to G.4 is:

```
SCHEMA artifact;

  TYPE gender = ENUMERATION OF (not-known, male, female);
  END_TYPE;

  TYPE general_approval = EXTENSIBLE ENUMERATION OF (approved, rejected);
  END_TYPE;

  -- other declarations dependent on gender and general_approval
  ...

END_SCHEMA; -- end artifact
```

The final ISO 10303-11:1994 schema is:

```
SCHEMA longform;

  TYPE gender = ENUMERATION OF (not-known, male, female);
  END_TYPE;

  TYPE general_approval = ENUMERATION OF (approved, rejected);
  END_TYPE;

  -- other declarations dependent on gender and general_approval
  ...

END_SCHEMA; -- end longform
```

EXAMPLE 2 The intermediate schema generated according to G.4 is:

```
SCHEMA artifact;

  TYPE general_approval = EXTENSIBLE ENUMERATION OF
       (approved, rejected);
  END_TYPE;

  TYPE domain2_approval = EXTENSIBLE ENUMERATION BASED_ON general_approval WITH
       (pending);
  END_TYPE;

  -- other declarations dependent on domain2_approval
  ...

END_SCHEMA; -- end artifact
```

The final ISO 10303-11:1994 schema is:

```
SCHEMA longform;

  TYPE general_approval = ENUMERATION OF
       (approved, rejected, pending);
  END_TYPE;
```


Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

```
  TYPE domain2_approval = general_approval;
  END_TYPE;

  -- other declarations dependent on domain2_approval
  ...

END_SCHEMA; -- end longform
```

EXAMPLE 3 The intermediate schema generated according to G.4 is:

```
SCHEMA artifact;

  TYPE general_approval = EXTENSIBLE ENUMERATION OF
       (approved, rejected);
  END_TYPE;

  TYPE domain2_approval = EXTENSIBLE ENUMERATION BASED_ON general_approval WITH
       (pending);
  END_TYPE;

  TYPE specific_approval = ENUMERATION BASED_ON domain2_approval WITH
       (rework);
  END_TYPE;

  TYPE domain3_approval = EXTENSIBLE ENUMERATION BASED_ON general_approval WITH
       (cancelled);
  END_TYPE;

  -- other declarations dependent on these types
  ...

END_SCHEMA; -- end artifact
```

The final ISO 10303-11:1994 schema is:

```
SCHEMA longform;

  TYPE general_approval = ENUMERATION OF
       (approved, rejected, pending, cancelled, rework);
  END_TYPE;

  TYPE domain2_approval = general_approval;
  WHERE
    wr1 : SELF <> cancelled;
  END_TYPE;

  TYPE specific_approval = domain2_approval;
  END_TYPE;

  TYPE domain3_approval = general_approval;
  WHERE
    wr1 : SELF <> pending;
    wr2 : SELF <> rework;
  END_TYPE;

  -- other declarations dependent on these types
  ...

END_SCHEMA; -- longform
```


©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Special attention needs to be given to the fact that enumeration data types according to ISO 10303-11:1994 of this document imply an order of the enumeration items. This concept of order is not present in ISO 10303-11:2003 .

G.5.3.2 Extensible select

Extensible selects (see 8.4.2) shall be converted according to the procedure above to select data types that are not extensible. The name of the source construct shall be maintained in the target.

A select data type that is based on an extensible select data type shall be converted to a defined data type that has as underlying data type the target select data type of the extensible select that it is based on. The name of the defined data type in the target model shall be the name of the corresponding select data type in the source. The defined data type shall exclude select items that are invalid in its context, but that are specified for its underlying select, from instantiation in the context schema by local rules. One local rule shall be created for each select item that shall be excluded.

EXAMPLE 1 The intermediate schema generated according to G.4 is:

```
SCHEMA artifact;

  TYPE attachment_method = EXTENSIBLE SELECT
       (nail);
  END_TYPE;

  ENTITY nail;
  END_ENTITY;

  TYPE permanent_attachment = SELECT BASED_ON attachment_method WITH
      (glue, weld);
  END_TYPE;

  TYPE simple_attachment = SELECT BASED_ON attachment_method WITH
       (needle, tape);
  END_TYPE;

  -- declarations of glue and others

END_SCHEMA; -- end artifact
```

The final ISO 10303-11:1994 schema is:

```
SCHEMA longform;

  TYPE attachment_method = SELECT
       (nail, glue, weld, needle, tape);
  END_TYPE;

  ENTITY nail;
  END_ENTITY;

  TYPE permanent_attachment = attachment_method;
  WHERE
    wr1 : NOT ('LONGFORM.NEEDLE' IN TYPEOF(SELF));
    wr2 : NOT ('LONGFORM.TAPE' IN TYPEOF(SELF));
  END_TYPE;
```


Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

```
  TYPE simple_attachment =  attachment_method;
  WHERE
    wr1 : NOT ('LONGFORM.GLUE' IN TYPEOF(SELF));
    wr2 : NOT ('LONGFORM.WELD' IN TYPEOF(SELF));
  END_TYPE;

-- declarations of glue and others

END_SCHEMA; -- end longform
```

EXAMPLE 2 The intermediate schema generated according to G.4 is called `problem`:

```
SCHEMA problem;

  ENTITY e1;
    attr : t1;
  END_ENTITY;

  ENTITY e2;
    attr : t3;
  END_ENTITY;

  TYPE t1 = SELECT
       (t2, t3);
  END_TYPE;

  TYPE t2 = INTEGER;
  END_TYPE;

  TYPE t3 = EXTENSIBLE SELECT;
  END_TYPE;

END_SCHEMA;
```

The `problem` schema is a referentially complete ISO 10303-11:2003 data specification. However, it is impossible to convert this to a specification conforming to ISO 10303-11:1994 because a SELECT data type cannot have an empty list of selectable items. Deleting `t3` does not solve the problem because it is referenced by the entity `e2`.

G.5.4 Conversion of subtype constraints

The keyword SUBTYPE_CONSTRAINT is not a part of ISO 10303-11:1994. SUBTYPE_CONSTRAINT declarations and are, therefore, eliminated by the conversion process. The semantics of the constraint, however, shall be maintained in the ISO 10303-11:1994 longform.

The conversions of TOTAL_OVER constraints and constraints on valid instantiations of subtype/supertype graphs are described in the following sub-clauses.

G.5.4.1 Total over constraint

The following conversion rules apply to TOTAL_OVER constraints:

— The longform schema shall maintain the semantics of the TOTAL_OVER constraint even if only one of the constituents of the constraint is interfaced.

— For each TOTAL_OVER constraint of a SUBTYPE_CONSTRAINT a global RULE shall be added

 ©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

to the SCHEMA.

— The RULE name shall be `total_over_<subtype constraint name>`.

— The RULE shall be valid FOR the supertype entity that the TOTAL_OVER was specified for.

— The WHERE rule in the global RULE shall ensure that each instance of the target supertype is of the data type(s) of the one or several subtypes that are specified in the source TOTAL_OVER constraint and that are interfaced into the target longform schema.

— Remarks with tags that relate to the SUBTYPE_CONSTRAINT shall be reassigned to the new RULE; if several global rules are created, the remark shall be repeated for each rule.

NOTE It is recommended that reassigned tagged remarks are manually edited at the end of the conversion process.

EXAMPLE The intermediate schema generated according to G.4 is:

```
SCHEMA artifact;

  ENTITY e1;
  END_ENTITY;

  ENTITY e2 SUBTYPE OF (e1);
  END_ENTITY;

  SUBTYPE_CONSTRAINT sc_total_over FOR e1;
    TOTAL_OVER (e2);
  END_SUBTYPE_CONSTRAINT;

END_SCHEMA; -- end artifact
```

The final ISO 10303-11:1994 schema is:

```
SCHEMA longform;

  ENTITY e1;
  END_ENTITY;

  ENTITY e2 SUBTYPE OF (e1);
  END_ENTITY;

  RULE total_over_sc_total_over FOR (e1);
  WHERE
    (*"total_over_sc_total_over.wr1" All instances of e1
      shall also be of entity data type e2. *)
   wr1 : SIZEOF (QUERY(e1_i <* e1 |
         SIZEOF (['LONGFORM.E2'] * TYPEOF(e1_i)) = 0)) = 0;
  END_RULE;

END_SCHEMA; -- end longform
```

G.5.4.2 Subtype/supertype instantiation constraints

If the SUBTYPE_CONSTRAINT includes constraints on which subtype/supertype graphs may be instantiated, these shall be moved into SUPERTYPE constraints. Such constraints typically in-

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

clude the keywords AND, ANDOR, and ONEOF. The SUPERTYPE constraints shall be built by the following rules:

a) For each entity constrained by a SUBTYPE_CONSTRAINT that contains a SUPERTYPE constraint, the SUPERTYPE constraint is enclosed in parentheses and added to a SUPERTYPE constraint specified in the entity.

b) Each constraint that results from a SUBTYPE_CONSTRAINT is combined via ANDOR with any other constraints added from different SUBTYPE_CONSTRAINT specifications.

Remarks with tags that relate to a SUBTYPE_CONSTRAINT with such constraints shall be reassigned to the entity with the SUPERTYPE constraint.

NOTE It is recommended that reassigned tagged remarks are manually edited at the end of the conversion process.

EXAMPLE The intermediate schema generated according to G.4 is:

```
SCHEMA artifact;

  ENTITY p;
  END_ENTITY;

  SUBTYPE_CONSTRAINT p_subs FOR p;
    m AND a;
  END_SUBTYPE_CONSTRAINT;

  ENTITY m SUBTYPE OF (p);
  END_ENTITY;

  ENTITY a ABSTRACT SUBTYPE OF (p);
  END_ENTITY;

  ENTITY l SUBTYPE OF (a);
  END_ENTITY;

END_SCHEMA; -- end artifact
```

The final ISO 10303-11:1994 schema is:

```
SCHEMA longform;

  ENTITY p
  SUPERTYPE OF (m AND a);
  END_ENTITY;

  ENTITY a ABSTRACT SUPERTYPE
    SUBTYPE OF (p);
  END_ENTITY;

  ENTITY l SUBTYPE OF (a);
  END_ENTITY;

  ENTITY m SUBTYPE OF (p);
  END_ENTITY;

END_SCHEMA; -- end longform
```


©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS

Not for Resale

G.5.5 Conversion of abstract entity and generalized types

ABSTRACT ENTITY declarations shall be converted into ABSTRACT SUPERTYPE constraints on the entity in the ISO 10303-11:1994 schema.

Formal parameters and local variables of functions and procedures that are declared to be of type GENERIC_ENTITY shall be converted to GENERIC. The function or procedure algorithm may need to be enhanced to ensure type compatibility with the GENERIC type.

An entity may have an attribute that is declared to be of a generalized data type, such as an AGGREGATE, a GENERIC_ENTITY, or a SELECT data type that is constrained to select items of type GENERIC_ENTITY. For such attributes in the ABSTRACT ENTITY data type that have a generalized type as their domain the following conversion rules apply:

— if all subtypes redeclare the attribute domain to be of the same type, migrate that type to be the type of the attribute in the SUPERTYPE in the ISO 10303-11:1994 schema;

— in all other cases, create a SELECT data type in the ISO 10303-11:1994 schema and add all named data types that are the types of the attribute in any redeclarations in any SUBTYPE of the SUPERTYPE. The name of the select data type shall be concatenated of the entity data type name of the ABSTRACT SUPERTYPE and the name of the attribute that will be of the select data type. The character '`_`' shall — without the quotes — be placed between the two names. The term '`_select`' shall be added to the name (see `binary_relationship_end_one_select` in the example below). Within the subtypes, this new supertype attribute shall be redeclared to be of the data type of the one select item that it was declared to be in the source schema.

  If the type of redeclared attribute results in an un-named data type, such as an aggregate, then defined data types are created to support the attribute type, and used in the SELECT data type. The rule for naming these supporting defined data types are: the name of the aggregation type concatenated with '`_of_`' and the name of the base data type for the aggregation (this may be recursive).

EXAMPLE In this example the intermediate schema resulting from the process in G.4 is called `abstract_example`.

```
SCHEMA abstract_example;

  ENTITY person;
  END_ENTITY;

  ENTITY product;
  END_ENTITY;

  ENTITY organization;
  END_ENTITY;

  ENTITY nary_relationship ABSTRACT;
    end_one : AGGREGATE OF GENERIC_ENTITY;
    end_two : GENERIC_ENTITY;
  END_ENTITY;

  ENTITY product_of_organization
    SUBTYPE OF (nary_relationship);
    SELF\nary_relationship.end_one: SET OF product;
```


Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

```
    SELF\nary_relationship.end_two: organization;
  END_ENTITY;

  ENTITY person_in_organization
    SUBTYPE OF (binary_relationship);
    SELF\nary_relationship.end_one: SET OF person;
    SELF\nary_relationship.end_two: organization;
  END_ENTITY;

END_SCHEMA; -- abstract_example
```

During the generation of the longform, the ABSTRACT construct is converted into a SUPERTYPE constraint. A SELECT data type is introduced to maintain the different data types that the generalized attribute in the supertype resulted in in the subtypes. Since the data types to be used in the select are not named data types, defined data types are created as required.

The final ISO 10303-11:1994 schema is:

```
SCHEMA longform;

  ENTITY person;
  END_ENTITY;

  ENTITY product;
  END_ENTITY;

  ENTITY organization;
  END_ENTITY;

  TYPE set_of_product = SET OF product;
  END_TYPE;

  TYPE set_of_person = SET OF person;
  END_TYPE;

  TYPE nary_relationship_end_one_select = SELECT
       (set_of_person, set_of_product);
  END_TYPE;

  ENTITY nary_relationship
    ABSTRACT SUPERTYPE;
    end_one : nary_relationship_end_one_select;
    end_two : organization;
  END_ENTITY;

  ENTITY product_of_organization
    SUBTYPE OF (nary_relationship);
    SELF\nary_relationship.end_one: SET OF product;
  END_ENTITY;

  ENTITY person_in_organization
    SUBTYPE OF (nary_relationship);
    SELF\nary_relationship.end_one: SET OF person;
  END_ENTITY;

END_SCHEMA; -- longform
```


©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS

Not for Resale

G.5.6 Conversion of attributes renamed in a redeclaration

For each attribute that is RENAMED in a redeclaration in a subtype, the following conversion rules apply:

— redeclarations that only change the name of the attribute, but do not change its data type shall be removed;

— in the remaining redeclarations, the keyword RENAMED and the subsequent new attribute name shall be removed;

— DERIVE attributes with the new names shall be created in the current SUBTYPE;

— the derivations of the attribute values shall be
`SELF\<supertype name>.<old attribute name>`.

EXAMPLE In this example the intermediate schema resulting from the process in G.4 is called `renamed_example`.

```
SCHEMA renamed_example;

  ENTITY binary_relationship;
    end_one : being;
    end_two : structure;
  END_ENTITY;

  ENTITY being;
  END_ENTITY;

  ENTITY structure;
  END_ENTITY;

  ENTITY person
    SUBTYPE OF (being);
  END_ENTITY;

  ENTITY person_in_structure
    SUBTYPE OF (binary_relationship);
     SELF\binary_relationship.end_one  RENAMED the_person    : person;
     (*"person_in_structure.end_two" The following attribute is only
       renamed, not specialized. *)
     SELF\binary_relationship.end_two  RENAMED the_structure : structure;
  END_ENTITY;

END_SCHEMA; -- end renamed_example
```

During the generation of the longform, the RENAMED constructs are converted into DERIVE attributes. Entity `person_in_structure` is modified as follows:

```
...
  ENTITY person_in_structure
    SUBTYPE OF (binary_relationship);
    SELF\binary_relationship.end_one : person;
  DERIVE
    the_person    : person    := SELF\binary_relationship.end_one;
    the_structure : structure := SELF\binary_relationship.end_two;
END_ENTITY;
```

©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

. . .

NOTE Failures may occur in functions where the attribute name following the RENAMED keyword has anything assigned to it. Manual manipulation of the schema will be required in such cases to prevent this.

©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS

Not for Resale

Annex H
(informative)
Relationships

H.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 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 (that is, 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 (that is, one instance of `second` may be related to zero or more instances of `first`); this is the default cardinality of the inverse relationship.

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

```
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:

$$\texttt{E.A} \xrightarrow{\{m:n\} \qquad \{p:q\}} \texttt{T}$$

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.

H.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:

$$\texttt{E.A} \xrightarrow{\{m:1\} \qquad \{p:q\}} \texttt{T}$$

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, $m = 1$;

 ©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

— 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:

EXAMPLE 1 CIRCLE.CENTRE $\underrightarrow{\quad \{1:1\} \quad\quad \{0:\ ?\} \quad}$ POINT

Every CIRCLE has exactly one POINT playing the role of CENTRE. Every POINT may play the role of CENTRE in any number of CIRCLEs (including none). This could be declared by:

```
ENTITY point;
  ...
END_ENTITY;

ENTITY circle;
  centre : point;
  ...
END_ENTITY;
```

EXAMPLE 2 PRODUCT_VERSION.BASE_PRODUCT $\underrightarrow{\quad \{1:1\} \quad\quad \{1:\ ?\} \quad}$ 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_VERSIONs, 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;
```

EXAMPLE 3 PERSON.LUNCH $\underrightarrow{\quad \{0:1\} \quad\quad \{0:\ ?\} \quad}$ 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;
```

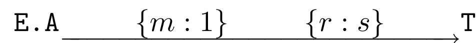

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

H.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 H.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:

EXAMPLE 1 `POLY_CURVE.COEF` $\xrightarrow{\{1:1\} \qquad \{0:\ ?\}}$ `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_CURVE`s (including none). This could be declared by:

```
ENTITY poly_curve;
  coef : LIST [1:?] OF REAL;
  ...
END_ENTITY;
```

EXAMPLE 2 `LOOP.EDGES` $\xrightarrow{\{0:1\} \qquad \{0:1\}}$ `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;
     ...
```


©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
END_ENTITY;
```

H.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:

$$\texttt{E.A} \xrightarrow{\quad \{k:l\} \qquad \{p:q\} \quad} \texttt{FUND(T)}$$

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:

EXAMPLE 1 Contrast this with example 1.

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

POLY_CURVE.COEF ——{1 : ?}——{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_CURVEs (including none). This could be declared by:

```
ENTITY poly_curve;
  coef : LIST [1:?] OF REAL;
  ...
END_ENTITY;
```

EXAMPLE 2 Compare this with example 2.

LOOP.EDGES ——{0 : ?}——{2 : 2}——→ EDGE

A LOOP may consist of any number of EDGEs (including none). An EDGE has to be used for exactly two different LOOPs. 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;
```

H.1.4 Inverse attribute

Every relationship established by an attribute has an implicit inverse relationship. By default, this relationship is effectively ignored, that is, it can not 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, that is, $q = 1$. This is equivalent to INVERSE I : SET [0:1] OF E FOR A;

 ©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

— 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.

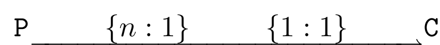
— if I is declared as INVERSE I : E FOR A, the effect is the same as if if had been declared as a SET [1:1] OF E. That is, every instance of R must play the role A in exactly one instance of E;

— 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.

H.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:



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:

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.

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Annex J
(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 ISO 10303.

J.1 Example single schema model

The model in the example 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 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;
  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;
```


©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
SUBTYPE_CONSTRAINT sc_person FOR person;
  ONEOF(female, male);
END_SUBTYPE_CONSTRAINT;

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;
  (*"years" This function calculates the number of years
     between the past date and the current date *)
END_FUNCTION;

END_SCHEMA;
```

J.2 Relationship sampler

The example 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 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;
```

©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

```
TYPE choice = SELECT
   (an_ent,
    name);
END_TYPE;

TYPE name = STRING;
END_TYPE;

TYPE strings = LIST [1:?] OF STRING;
END_TYPE;

END_SCHEMA;
```

J.3 Simple subtype/supertype tree

EXPRESS enables very complex subtype/supertype trees (and networks) to be defined. The tree shown in the example is relatively simple.

EXAMPLE Subtype/supertype tree in EXPRESS

```
SCHEMA simple_trees;

ENTITY super;
END_ENTITY;

ENTITY sub1
  SUBTYPE OF (super);
END_ENTITY;

ENTITY sub2;
  SUBTYPE OF (super);
END_ENTITY;

SUBTYPE_CONSTRAINT sc_sub2 FOR sub2;
  ABSTRACT;
  ONEOF(sub3, sub4);
END_SUBTYPE_CONSTRAINT;

ENTITY sub3
  SUBTYPE OF (sub2);
END_ENTITY;

ENTITY sub4
  SUBTYPE OF (sub2);
END_ENTITY;

ENTITY sub5
  SUBTYPE OF (super);
END_ENTITY;

END_SCHEMA;
```

J.4 Attribute redeclaration

EXPRESS permits the redeclaration of inherited attributes, provided the new attribute types are compatible. The example shows some of the permissable forms of redeclaration:

 ©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

— 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 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;
```

J.5 Multi-schema models

EXPRESS models consist of at least one schema. Example 1 shows a model that consists of two schemas.

EXAMPLE 1 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;
```


Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

```
    loc    : surface;
  END_ENTITY;

  ENTITY loop;
  END_ENTITY;

  SUBTYPE_CONSTRAINT sc_loop FOR loop;
    ABSTRACT;
    ONEOF(eloop, vloop);
  END_SUBTYPE_CONSTRAINT;

  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 2. 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 2 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;
```

 ©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
END_SCHEMA;

SCHEMA whosits;
  REFERENCE FROM stuff;
  REFERENCE FROM whatsits;
END_SCHEMA;
```


Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Annex K
(informative)
Deprecated features of EXPRESS

In this edition of EXPRESS, there are several concepts that now have two different syntaxes with which they may be specified. This duplication exists so that existing schemas remain valid while allowing for the use of new constructs. In future editions of EXPRESS, this duplication will be eliminated by the removal of the EXPRESS edition 1 syntax for constructs for which new syntax has been made available. For this reason, the use of the following syntax is deprecated:

— the ABSTRACT SUPERTYPE denoted by (ABS) in the entity in EXPRESS-G diagrams;

— the SUPERTYPE constraint specification syntax (see 9.2.3.2 and 9.2.5) which may now be specified by the new SUBTYPE_CONSTRAINT declaration.

There are also semantics associated with concepts that exist in EXPRESS edition 1 that are modified or removed in this edition of EXPRESS. For this reason, the following semantics are deprecated:

— the ordering of the enumeration items associated with a non-extensible, non-extension ENUMERATION data type.

©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Annex L
(informative)
Examples of the new EXPRESS constructs

This annex provides examples of the new constructs that were added to the language by this edition of this part of ISO 10303. No claim is made concerning the quality of the model below. It is simply showing examples of how the new constructs may be used.

L.1 Product management example

The example shows the use of the following constructs in a schema that is a resource to be used by schemas that are more complete with respect to the application domain:

— extensible constructed data types;

— GENERIC_ENTITY constraint on SELECT;

— ABSTRACT ENTITY;

— renaming of attributes;

— SUBTYPE_CONSTRAINT.

EXAMPLE This example uses the extensible constructed data types.

```
SCHEMA my_product_management;

USE FROM generic_product_management;

TYPE my_additional_categories = ENUMERATION BASED_ON product_category_names WITH
  ( document, drawing, electromechanical, mechanical, electrical, pump );
END_TYPE;

TYPE my_additional_values = ENUMERATION BASED_ON approval_status_values WITH
  ( approved, disapproved, pending );
END_TYPE;

TYPE my_approvable_objects = EXTENSIBLE SELECT BASED_ON approvable_objects WITH
  ( product, product_category, product_to_category_relationship );
END_TYPE;

ENTITY approval_by_person_in_organization
  SUBTYPE OF ( approval );
  SELF\approval.approved_by : person_in_organization_relationship;
END_ENTITY;

ENTITY approval_by_person
  SUBTYPE OF ( approval );
  SELF\approval.approved_by : person;
END_ENTITY;

SUBTYPE_CONSTRAINT not_both FOR approval;
  ONEOF ( approval_by_person, approval_by_person_in_organization );
END_SUBTYPE_CONSTRAINT;

END_SCHEMA;
```


Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

```
SCHEMA generic_product_management;

TYPE product_category_names = EXTENSIBLE ENUMERATION OF ( part, tool, raw_material );
END_TYPE;

TYPE approval_status_values = EXTENSIBLE ENUMERATION;
END_TYPE;

TYPE approvable_objects = EXTENSIBLE GENERIC_ENTITY SELECT;
END_TYPE;

ENTITY product;
  name : STRING;
END_ENTITY;

ENTITY product_category;
  name : product_category_names;
END_ENTITY;

ENTITY binary_entity_relationship ABSTRACT;
  end_one : GENERIC_ENTITY;
  end_two : GENERIC_ENTITY;
END_ENTITY;

ENTITY product_to_category_relationship
  SUBTYPE OF ( binary_entity_relationship );
  SELF\binary_entity_relationship.end_one RENAMED the_category : product_category;
  SELF\binary_entity_relationship.end_two RENAMED the_product  : product;
END_ENTITY;

ENTITY approval ABSTRACT;
  approved_by    : GENERIC_ENTITY;
  status         : approval_status_values;
  approved_items : SET[1:?] OF approvable_objects;
END_ENTITY;

ENTITY person;
  name : STRING;
END_ENTITY;

ENTITY organization;
  name : STRING;
END_ENTITY;

ENTITY person_in_organization_relationship
  SUBTYPE OF ( binary_entity_relationship );
  role_of_person : STRING;
  SELF\binary_entity_relationship.end_one RENAMED the_person       : person;
  SELF\binary_entity_relationship.end_two RENAMED the_organization : organization;
END_ENTITY;

END_SCHEMA;
```


©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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.

[4] SANDERSON, D. *The Proposed Amendment to EXPRESS - Its motivation, features and relationship to EXPRESS Edition 2*

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Index

? (constant) . 24–28, 39, 42, 47, 48, 60, 71, 72, 93, 95, 97, 98, 100–102, 104, 106–112, 114, 115, 117, 118, 124–127, 129, 130, 132, 133, 135, 138, 141, 142, 144–149, 235
abs (function) 15, 133
abstract (reserved word) 14, 55, 76, 195, 196, 205, 232, 241, 248, 249
abstract entity (reserved word) 35, 54, 55, 66, 68, 70, 195, 199, 206, 231
abstract entity: EXPRESS-G symbol 195
abstract supertype (reserved word) 49, 55, 56, 75, 169, 195, 196, 199, 206, 231
abstract supertype: EXPRESS-G symbol 194
acos (function) 15, 133
aggregate (reserved word) 14, 23, 35, 61, 65, 66, 69, 70, 117, 192, 231
alias (reserved word) 14, 81, 121, 122
and (reserved word) 14, 56, 58, 59, 77, 93, 104, 105, 121, 167, 168, 170, 173, 176, 180–182, 195, 230
and: EXPRESS-G symbol 195
andor (reserved word) 14, 34, 56, 58, 59, 77, 167–170, 181, 183, 230
array (reserved word) 14, 23, 24, 62, 66, 70, 71, 107, 111, 117, 124, 138, 139, 141, 143–145, 192
as (reserved word) 14, 87, 181
asin (function) 15, 133
assignment compatibility 123
atan (function) 15, 134
attribute: EXPRESS-G symbol 193
bag (reserved word) ... 14, 23, 26, 43, 44, 62, 66, 70, 71, 108, 111, 117, 138, 139, 141, 143–145, 147, 192, 239, 241
based_on (reserved word) .... xii, 14, 30, 34, 90, 145, 146
begin (reserved word) 14, 127
binary (reserved word) 14, 20, 22, 23, 62, 92, 103, 124, 125, 144
blength (function) 15, 96, 134
boolean (reserved word) 14, 20, 21, 61, 96, 132, 133, 144
by (reserved word) 14
cardinality 191–193, 235
case (reserved word) 14, 121, 126
character set 10
const_e (constant) 15, 132
constant (reserved word) 14, 63
constraint: EXPRESS-G symbol 189, 192
constructed data types: EXPRESS-G symbol 187, 193
cos (function) 15, 134
defined data type: EXPRESS-G symbol 188, 193
derive (reserved word) 14, 42, 206, 233
derive: EXPRESS-G symbol 193
diagram: abstraction 200
diagram: complete 198–200
diagram: entity level 191, 199
diagram: schema level 198, 200
div (reserved word) 14, 93
else (reserved word) 14, 127
end (reserved word) 14, 127
end_alias (reserved word) 14, 81
end_case (reserved word) 14
end_constant (reserved word) 14
end_entity 14, 81

 ©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS

Not for Resale

end_function (reserved word) 14, 82
end_if (reserved word) 14
end_local (reserved word) 14
end_procedure (reserved word) 14, 83
end_repeat (reserved word) 14, 84, 127, 131
end_rule (reserved word) 14, 84
end_schema (reserved word) 14, 85
end_subtype_constraint (reserved word) .... xii, 14, 86
end_type (reserved word) 14, 86
entity (reserved word) ... 14, 28, 34, 35, 40, 41, 50–52, 54, 55, 61, 62, 70, 75, 81, 147, 169, 188, 193, 206, 208, 217, 249
entity data type: EXPRESS-G symbol 188, 193
enumeration (reserved word) 14, 29–31, 112, 187, 193, 205, 224, 248
enumeration: EXPRESS-G symbol 187, 193
escape (reserved word) 14, 121, 127
exists (function) 15, 42, 135
exp (function) 15, 135
EXPRESS character set 10
extensible (reserved word) .... xii, 14, 30, 31, 33–35, 61, 205, 224
extensible constructed data types: EXPRESS-G symbol 188
extension relationship: EXPRESS-G symbol 193
false (constant) ... 15, 19, 21, 28, 39, 47, 72, 92, 95–102, 105, 106, 110, 111, 126, 127, 129, 130, 132, 135, 141, 148, 149
fixed (reserved word) 14, 22, 23, 62
for (reserved word) 14, 45, 122, 169, 229
format (function) 15, 135
from (reserved word) 14, 181
function (reserved word) 13, 14, 64, 65, 69, 82, 121, 131, 188, 217
function: EXPRESS-G symbol 188
generalized data types: EXPRESS-G symbol 186
generic (reserved word) 14, 35, 42, 43, 45, 65, 67, 69–71, 97, 117, 231
generic-entity: EXPRESS-G symbol 187
generic_entity (reserved word) xii, 14, 34, 35, 52, 61, 65, 66, 68, 69, 186, 187, 200, 206, 231, 249
hibound (function) 15, 138
hiindex (function) 15, 70, 138
if (reserved word) 14, 121, 127, 128
in (reserved word) 14, 93, 94, 100, 101, 107–111
indeterminate value 132
inheritance relationship: EXPRESS-G symbol 189, 194
insert (procedure) 15, 128, 149
integer (reserved word) 14, 20, 21, 51, 61, 93, 144, 149, 150
inverse (reserved word) 14, 43, 235, 237, 239, 240
inverse: EXPRESS-G symbol 194
length (function) 15, 96, 139
like (reserved word) 14, 93, 94, 102
line styles: EXPRESS-G 185, 189
list (reserved word) . 14, 23, 25, 62, 66, 70, 71, 107, 111, 117, 124, 125, 138, 139, 141, 143–145, 149, 150, 192, 239
lobound (function) 15, 139
local (reserved word) 14, 71
log (function) 15, 140
log10 (function) 15

©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

log2 (function) ... 15
logical (reserved word) 14, 20, 21, 61, 72, 92, 95, 96, 100–102, 110, 111, 127, 130–133, 144, 148, 149
loindex (function) ... 15, 70, 140, 141
mod (reserved word) ... 14, 93
non-select data type ... 120
not (reserved word) ... 14, 93, 104, 105, 120
notation ... 9
number (reserved word) ... 14, 20, 51, 61, 93, 144, 197
nvl (function) ... 15, 42, 48, 141
odd (function) ... 15, 141
of (reserved word) ... 14, 90, 117
oneof (reserved word) .. 14, 56, 57, 59, 76, 167, 168, 170, 172–175, 179–182, 195, 196, 199, 200, 230
oneof: EXPRESS-G symbol ... 195
optional (reserved word) ... 14, 24, 41, 42, 46, 60, 112, 135, 189, 236, 238
optional attribute: EXPRESS-G symbol ... 189
or (reserved word) ... 14, 93, 104, 105
otherwise (reserved word) ... 14, 126
page reference: EXPRESS-G symbol ... 190, 198
pi (constant) ... 15, 132
procedure (reserved word) ... 14, 65, 69, 83, 121, 131, 188, 217
procedure: EXPRESS-G symbol ... 188
query (reserved word) ... 14, 83, 84, 107, 111, 120, 147
real (reserved word) ... 14, 20, 21, 29, 51, 61, 62, 93, 94, 132, 144, 197, 238, 240
redeclared attribute ... 41, 50, 51, 60, 61
redeclared attribute: EXPRESS-G symbol ... 194, 196
reference ... 198
reference (reserved word) . 14, 86–89, 142, 146, 181, 191, 198, 200, 204, 205, 207, 209, 212, 217
reference: EXPRESS-G symbol ... 198
relationship: EXPRESS-G symbol ... 189, 193
remove (procedure) ... 15, 150
rename in interface: EXPRESS-G symbol ... 191, 198
renamed (reserved word) ... xii, 14, 52, 206, 233, 234
renamed: EXPRESS-G symbol ... 194, 196
repeat (reserved word) ... 14, 84, 108, 121, 127–131
return (reserved word) ... 14, 64, 121, 131
rolesof (function) ... 15, 87, 142
rule (reserved word) ... 14, 65, 72, 73, 84, 121, 189, 192, 199, 205, 206, 217, 219, 220, 228, 229
rule: EXPRESS-G symbol ... 189
rules: EXPRESS-G symbol ... 189
schema (reserved word) ... 13, 14, 62, 85, 189, 204–207, 223, 229
schema interface ... 198
schema reference: EXPRESS-G symbol ... 190, 191, 197
schema-schema relationship: EXPRESS-G symbol ... 189
schema: EXPRESS-G symbol ... 189
scope ... 50, 78
select (reserved word) 14, 29, 33–36, 38, 61, 90, 91, 120, 145, 187, 192, 193, 200, 205, 206, 217, 218, 224, 228, 231, 232, 249
select: EXPRESS-G symbol ... 187, 193
self (constant) ... 15, 39, 42, 43, 47, 132

 ©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

set (reserved word) 14, 23, 26, 27, 43, 62, 66, 70, 71, 108, 111, 117, 138, 139, 141, 143–146, 192, 241
simple data types: EXPRESS-G symbol . . . 186
sin (function) . . . 15, 143
sizeof (function) . . . 15, 109, 143, 147
skip (reserved word) . . . 14, 121, 131
specialization . . . 38, 48, 61
sqrt (function) . . . 15, 143
string (reserved word) . . . 14, 20–22, 39, 62, 92, 106, 124, 125, 144, 147
subtype (reserved word) . . . 14, 49, 217, 231, 233
subtype-constraint: EXPRESS-G symbol . . . 188, 195
subtype_constraint (reserved word) xii, 14, 55, 56, 74–77, 86, 169, 170, 188, 189, 195, 205, 220, 222, 228–230, 248, 249
supertype (reserved word) . . . 14, 55, 90, 205, 217, 229–232, 248
symbol . . . 185
tan (function) . . . 15, 144
then (reserved word) . . . 14, 127
to (reserved word) . . . 14
total-over: EXPRESS-G symbol . . . 195
total_over (reserved word) . . . xii, 14, 74–76, 168, 172, 173, 181, 195, 200, 206, 219, 228, 229
true (constant) . . . 15, 19, 21, 28, 39, 47, 72, 92, 95–102, 105, 106, 110–112, 127–130, 133, 135, 141, 146, 148, 149
type (reserved word) . . . 14, 29, 39, 86, 188, 206
typeof (function) . . . 15, 144, 146
unique (reserved word) . . . 14, 24, 25, 44, 46, 100, 192
unknown (constant) . 15, 19, 21, 25, 28, 39, 47, 48, 72, 92, 95–102, 104–106, 110, 111, 126, 127, 129, 130, 133, 148, 149
until (reserved word) . . . 14, 129–131
use (reserved word) . . . 14, 85–89, 142, 146, 181, 182, 184, 191, 198, 200, 204, 205, 207, 217
use: EXPRESS-G symbol . . . 198
usedin (function) . . . 15, 147
value (function) . . . 15, 148
value_in (function) . . . 15, 97, 101, 148
value_unique (function) . . . 15, 27, 149
var (reserved word) . . . 14, 64, 65
visibility . . . 78, 79
where (reserved word) . . . 14, 39, 47, 52, 121, 192, 224, 229
while (reserved word) . . . 14, 129, 130
with (reserved word) . . . xii, 14
xor (reserved word) . . . 14, 93, 104, 105

©ISO 2004 — All rights reserved

Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

ICS 25.040.40

Price based on 255 pages

© ISO 2004 – All rights reserved
Copyright International Organization for Standardization
Reproduced by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale