

**Implementation of
ISO 10303-11 : 1994**

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

**Part 11. Description methods: The
EXPRESS language reference manual**

UDC 65.011.56:681.3:519.682

Committees responsible for this British Standard

The preparation of this British Standard was entrusted to Technical Committee ATM/4, Product data exchange, upon which the following bodies were represented:

- British Aerospace
- CIMIO Ltd.
- Construction Industry Computing Association
- Department of Trade and Industry (Services Management Division (IMS))
- Institution of Chemical Engineers
- Institution of Electrical Engineers
- Institute of Materials
- Lloyd's Register of Shipping
- Ministry of Defence
- NAFEMS
- Nine Tiles Computer Systems Ltd.
- Radan Computational Limited
- Rutherford Appleton Laboratory
- South Bank University
- University of Bristol
- University of Leeds Industrial Services Ltd.
- University of Loughborough

This British Standard, having been prepared under the direction of the Advanced Manufacturing Technology Assembly, was published under the authority of the Standards Board and comes into effect on 15 September 1995

© BSI 1995

Amendments issued since publication

Amd. No.	Date	Text affected

The following BSI references relate to the work on this standard:
Committee reference AMT/4
Draft for comment 92/96731 DC

ISBN 0 580 24671 X

National foreword

This British Standard reproduces verbatim ISO 10303-11 : 1994 and implements it as the UK national standard.

This British Standard is published under the direction of the Advanced Manufacturing Technology Assembly whose Technical Committee AMT/4 has the responsibility to:

- aid enquirers to understand the text;
- present to the responsible international committee any enquiries on interpretation, or proposals for change, and keep UK interests informed;
- monitor related international and European developments and promulgate them in the UK.

NOTE. International and European Standards, as well as overseas standards, are available from Customer Services, BSI, 389 Chiswick High Road, London W4 4AL.

Compliance with a British Standard does not of itself confer immunity from legal obligations.

INTERNATIONAL STANDARD

ISO
10303-11

First edition
1994-12-15

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 10303-11:1994(E)

AVA Documents
Document Sharing Hub

Contents	Page
1 Scope	1
2 Normative references	1
3 Definitions	2
3.1 Terms defined in ISO 10303-1	2
3.2 Other definitions	2
3.2.1 complex entity data type	2
3.2.2 complex entity (data type) instance	2
3.2.3 constant	2
3.2.4 data type	2
3.2.5 entity	3
3.2.6 entity data type	3
3.2.7 entity (data type) instance	3
3.2.8 instance	3
3.2.9 partial complex entity data type	3
3.2.10 partial complex entity value	3
3.2.11 population	3
3.2.12 simple entity (data type) instance	3
3.2.13 subtype/supertype graph	3
3.2.14 token	3
3.2.15 value	3
4 Conformance requirements	3
4.1 Formal specifications written in EXPRESS	3
4.1.1 Lexical language	3
4.1.2 Graphical form	4
4.2 Implementations of EXPRESS	5
4.2.1 EXPRESS language parser	5
4.2.2 Graphical editing tool	5
5 Fundamental principles	5
6 Language specification syntax	6

© ISO 1994

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 the publisher.

International Organization for Standardization
Case Postale 56 • CH-1211 Genève 20 • Switzerland

Printed in Switzerland

6.1	The syntax of the specification	7
6.2	Special character notation	8
7	Basic language elements	8
7.1	Character set	9
7.1.1	Digits	9
7.1.2	Letters	9
7.1.3	Special characters	10
7.1.4	Underscore	10
7.1.5	Whitespace	10
7.1.6	Remarks	11
7.2	Reserved words	12
7.2.1	Keywords	12
7.2.2	Reserved words which are operators	13
7.2.3	Built-in constants	13
7.2.4	Built-in functions	13
7.2.5	Built-in procedures	13
7.3	Symbols	14
7.4	Identifiers	14
7.5	Literals	15
7.5.1	Binary literal	15
7.5.2	Integer literal	15
7.5.3	Real literal	16
7.5.4	String literal	16
7.5.5	Logical literal	18
8	Data types	18
8.1	Simple data types	18
8.1.1	Number data type	18
8.1.2	Real data type	19
8.1.3	Integer data type	19
8.1.4	Logical data type	20
8.1.5	Boolean data type	20
8.1.6	String data type	20
8.1.7	Binary data type	21
8.2	Aggregation data types	22
8.2.1	Array data type	23
8.2.2	List data type	24
8.2.3	Bag data type	25
8.2.4	Set data type	25
8.2.5	Value uniqueness on aggregates	26
8.3	Named data types	28
8.3.1	Entity data type	28
8.3.2	Defined data type	28
8.4	Constructed data types	29
8.4.1	Enumeration data type	29

ISO 10303-11:1994(E)

8.4.2	Select data type	30
8.5	Generalized data types	31
8.6	Data type usage classification	31
8.6.1	Base data types	32
8.6.2	Parameter data types	32
8.6.3	Underlying data types	32
9	Declarations	33
9.1	Type declaration	33
9.2	Entity declaration	35
9.2.1	Attributes	35
9.2.2	Local rules	40
9.2.3	Subtypes and supertypes	43
9.2.4	Subtype/supertype constraints	49
9.2.5	Implicit declarations	53
9.2.6	Specialization	55
9.3	Schema	55
9.4	Constant	56
9.5	Algorithms	56
9.5.1	Function	57
9.5.2	Procedure	58
9.5.3	Parameters	58
9.5.4	Local variables	62
9.6	Rule	63
10	Scope and visibility	65
10.1	Scope rules	67
10.2	Visibility rules	67
10.2.1	General rules of visibility	67
10.2.2	Named data type identifier visibility rules	68
10.3	Explicit item rules	69
10.3.1	Alias statement	69
10.3.2	Attribute	69
10.3.3	Constant	70
10.3.4	Enumeration item	70
10.3.5	Entity	70
10.3.6	Function	71
10.3.7	Parameter	72
10.3.8	Procedure	72
10.3.9	Query expression	73
10.3.10	Repeat statement	73
10.3.11	Rule	73
10.3.12	Rule label	74
10.3.13	Schema	74
10.3.14	Type	75
10.3.15	Type label	76

10.3.16	Variable	76
11	Interface specification	76
11.1	Use interface specification	77
11.2	Reference interface specification	77
11.3	The interaction of use and reference	78
11.4	Implicit interfaces	78
11.4.1	Constant interfaces	79
11.4.2	Defined data type interfaces	79
11.4.3	Entity data type interfaces	80
11.4.4	Function interfaces	81
11.4.5	Procedure interfaces	81
11.4.6	Rule interfaces	81
12	Expression	81
12.1	Arithmetic operators	83
12.2	Relational operators	85
12.2.1	Value comparison operators	85
12.2.2	Instance comparison operators	89
12.2.3	Membership operator	91
12.2.4	Interval expressions	92
12.2.5	Like operator	93
12.3	Binary operators	94
12.3.1	Binary indexing	94
12.3.2	Binary concatenation operator	95
12.4	Logical operators	95
12.4.1	NOT operator	95
12.4.2	AND operator	96
12.4.3	OR operator	96
12.4.4	XOR operator	96
12.5	String operators	96
12.5.1	String indexing	96
12.5.2	String concatenation operator	97
12.6	Aggregate operators	97
12.6.1	Aggregate indexing	98
12.6.2	Intersection operator	99
12.6.3	Union operator	99
12.6.4	Difference operator	100
12.6.5	Subset operator	102
12.6.6	Superset operator	102
12.6.7	Query expression	103
12.7	References	104
12.7.1	Simple references	104
12.7.2	Prefixed references	105
12.7.3	Attribute references	106
12.7.4	Group references	106

ISO 10303-11:1994(E)

12.8	Function call	108
12.9	Aggregate initializer	109
12.10	Complex entity instance construction operator	110
12.11	Type compatibility	111
13	Executable statements	112
13.1	Null (statement)	112
13.2	Alias statement	113
13.3	Assignment statement	113
13.4	Case statement	114
13.5	Compound statement	115
13.6	Escape statement	116
13.7	If ...Then ...Else statement	116
13.8	Procedure call statement	117
13.9	Repeat statement	117
13.9.1	Increment control	118
13.9.2	While control	119
13.9.3	Until control	119
13.10	Return statement	120
13.11	Skip statement	120
14	Built-in constants	121
14.1	Constant e	121
14.2	Indeterminate	121
14.3	False	121
14.4	Pi	121
14.5	Self	122
14.6	True	122
14.7	Unknown	122
15	Built-in functions	122
15.1	Abs - arithmetic function	122
15.2	ACos - arithmetic function	122
15.3	ASin - arithmetic function	123
15.4	ATan - arithmetic function	123
15.5	BLength - binary function	123
15.6	Cos - arithmetic function	123
15.7	Exists - general function	124
15.8	Exp - arithmetic function	124
15.9	Format - general function	124
15.9.1	Symbolic representation	125
15.9.2	Picture representation	126
15.9.3	Standard representation	127
15.10	HiBound - arithmetic function	127
15.11	HiIndex - arithmetic function	127
15.12	Length - string function	128
15.13	LoBound - arithmetic function	128

15.14	Log - arithmetic function	129
15.15	Log2 - arithmetic function	129
15.16	Log10 - arithmetic function	129
15.17	LoIndex - arithmetic function	130
15.18	NVL - null value function	130
15.19	Odd - arithmetic function	130
15.20	RolesOf - general function	131
15.21	Sin - arithmetic function	132
15.22	SizeOf - aggregate function	132
15.23	Sqrt - arithmetic function	133
15.24	Tan - arithmetic function	133
15.25	TypeOf - general function	133
15.26	UsedIn - general function	135
15.27	Value - arithmetic function	136
15.28	Value_in - membership function	137
15.29	Value_unique - uniqueness function	137
16	Built-in procedures	138
16.1	Insert	138
16.2	Remove	138

Annexes

A	EXPRESS language syntax	139
A.1	Tokens	139
A.1.1	Keywords	139
A.1.2	Character classes	142
A.1.3	Lexical elements	142
A.1.4	Remarks	143
A.1.5	Interpreted identifiers	143
A.2	Grammar rules	143
A.3	Cross reference listing	147
B	Determination of the allowed entity instantiations	155
B.1	Formal approach	155
B.2	Supertype operators	157
B.2.1	ONEOF	157
B.2.2	AND	157
B.2.3	ANDOR	157
B.2.4	Precedence of operators	157
B.3	Interpreting the possible complex entity data types	157
C	Instance limits imposed by the interface specification	168
D	EXPRESS-G: A graphical subset of EXPRESS	172
D.1	Introduction and overview	172
D.2	Definition symbols	172

ISO 10303-11:1994(E)

D.2.1	Symbol for simple data types	174
D.2.2	Symbols for constructed data types	174
D.2.3	Symbols for defined data types	175
D.2.4	Symbols for entity data types	175
D.2.5	Symbols for functions and procedures	175
D.2.6	Symbols for rules	175
D.2.7	Symbols for schemas	176
D.3	Relationship symbols	176
D.4	Composition symbols	177
D.4.1	Page references	178
D.4.2	Inter-schema references	178
D.5	Entity level diagrams	179
D.5.1	Role names	179
D.5.2	Cardinalities	179
D.5.3	Constraints	179
D.5.4	Constructed and defined data types	180
D.5.5	Entity data types	180
D.5.6	Inter-schema references	182
D.6	Schema level diagrams	183
D.7	Complete EXPRESS-G diagrams	184
D.7.1	Complete entity level diagram	184
D.7.2	Complete schema level diagram	185
E	Protocol implementation conformance statement (PICS)	187
E.1	EXPRESS language parser	187
E.2	EXPRESS-G editing tool	187
F	Information object registration	189
G	Relationships	190
G.1	Relationships via attributes	190
G.1.1	Simple relationship	191
G.1.2	Collective relationship	193
G.1.3	Distributive relationship	194
G.1.4	Inverse attribute	196
G.2	Subtype/supertype relationships	197
H	EXPRESS models for EXPRESS-G illustrative examples	198
H.1	Example single schema model	198
H.2	Relationship sampler	199
H.3	Simple subtype/supertype tree	200
H.4	Attribute redeclaration	201
H.5	Multi-schema models	201
J	Bibliography	204
Index		205

Figures

B.1	EXPRESS-G diagram of schema for example 155.	160
B.2	EXPRESS-G diagram of schema for example 156.	162
B.3	EXPRESS-G diagram of schema for example 157.	164
D.1	Complete entity level diagram of example 171 (Page 1 of 2).	173
D.2	Complete entity level diagram of example 171 (Page 2 of 2).	173
D.3	Symbols for EXPRESS simple data types.	174
D.4	Symbols for EXPRESS constructed data types.	174
D.5	Abbreviated symbols for the EXPRESS constructed data types when used as the representation of defined data types.	175
D.6	Example of alternative methods for representing an ENUMERATION	175
D.7	Symbols for EXPRESS defined data type.	175
D.8	Symbol for an EXPRESS entity data type.	176
D.9	Symbol for a schema.	176
D.10	Relationship line styles	176
D.11	Partial entity level diagram illustrating relationship directions from example 172. (Page 1 of 1)	177
D.12	Composition symbols: page references	178
D.13	Composition symbols: inter-schema references	178
D.14	Complete entity level diagram of example 172. (Page 1 of 1)	180
D.15	Complete entity level diagram of the inheritance graph from example 173. (Page 1 of 1)	182
D.16	Complete entity level diagram of example 174 showing attribute redeclarations in subtypes. (Page 1 of 1)	182
D.17	Complete entity level diagram of the top schema of example 175 illustrating inter- schema references. (Page 1 of 1).	183
D.18	Complete schema level diagram of example 175. (Page 1 of 1)	184
D.19	Complete schema level diagram of example 176. (Page 1 of 1)	184

Tables

1	EXPRESS keywords	12
2	EXPRESS reserved words which are operators	13

ISO 10303-11:1994(E)

3	EXPRESS reserved words which are constants	13
4	EXPRESS reserved words which are function names	13
5	EXPRESS reserved words which are procedure names	14
6	EXPRESS symbols	14
7	The use of data types	32
8	Supertype expression operator precedence	53
9	Scope and identifier defining items	66
10	Operator precedence	83
11	Pattern matching characters	94
12	NOT operator	96
13	AND operator	97
14	OR operator	98
15	XOR operator	98
16	Intersection operator – operand and result types	100
17	Union operator – operand and result types	101
18	Difference operator – operand and result types	102
19	Subset and superset operators - operand types	102
20	Example symbolic formatting effects	126
21	Picture formatting characters	126
22	Example picture formatting effects	127

Foreword

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

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

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

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

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

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

- Part 1, Overview and fundamental principles;
- Part 11, Description methods: The *EXPRESS* language reference manual;
- Part 21, Implementation methods: Clear text encoding of the exchange structure;
- Part 22, Implementation methods: Standard data access interface specification;
- Part 31, Conformance testing methodology and framework: General concepts;
- Part 32, Conformance testing methodology and framework: Requirements on testing laboratories and clients;
- Part 41, Integrated generic resources: Fundamentals of product description and support;
- Part 42, Integrated generic resources: Geometric and topological representation;
- Part 43, Integrated generic resources: Representation structures;
- Part 44, Integrated generic resources: Product structure configuration;
- Part 45, Integrated generic resources: Materials;
- Part 46, Integrated generic resources: Visual presentation;

ISO 10303-11:1994(E)

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

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

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

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

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

Introduction

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

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

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

Language overview

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

- the size and complexity of ISO 10303 demands that the language be parsable by both computers and humans. Expressing the information elements of ISO 10303 in a less formal manner would eliminate the possibility of employing computer automation in checking for inconsistencies in presentation or for creating any number of secondary views, including implementation views;
- the language is designed to enable partitioning of the diverse material addressed by ISO 10303. The schema is the basis for partitioning and intercommunication;
- the language focuses on the definition of entities, which represent objects of interest. The definition of an entity is in terms of its properties, which are characterized by specification of a domain and the constraints on that domain;
- the language seeks to avoid, as far as possible, specific implementation views. However, it is possible to manufacture implementation views (such as static file exchange) in an automatic and straightforward manner.

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

NOTES

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

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

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

1 Scope

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

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

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

The following are within the scope:

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

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

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

EXPRESS is not a programming language.

2 Normative references

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

ISO 10303-11:1994(E)

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

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

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

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

3 Definitions

3.1 Terms defined in ISO 10303-1

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

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

3.2 Other definitions

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

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

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

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

3.2.4 data type: a domain of values.

¹⁾To be published.

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

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

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

3.2.8 instance: a named value.

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

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

3.2.11 population: a collection of entity data type instances.

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

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

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

3.2.15 value: a unit of data.

4 Conformance requirements

4.1 Formal specifications written in EXPRESS

4.1.1 Lexical language

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

Levels of checking

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

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

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

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

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

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

4.1.2 Graphical form

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

Levels of checking

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

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

4.2 Implementations of EXPRESS

4.2.1 EXPRESS language parser

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

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

4.2.2 Graphical editing tool

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

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

5 Fundamental principles

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

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

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

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

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

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

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

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

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

6 Language specification syntax

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

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

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

6.1 The syntax of the specification

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

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

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

```

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

```

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

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

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

– The semantics of the enclosing braces are defined below:

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

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

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

Syntax:

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

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

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

- a) `string`
- b) `string ( 22 )`
- c) `string ( 19 ) fixed`

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

6.2 Special character notation

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

- `\a` represents characters in cells 21-7E of row 00, plane 00, group 00 of ISO 10646;
- `\n` represents a newline (system dependent) (see 7.1.5.2);
- `\q` is the quote (apostrophe) (') character and is contained within `\a`;
- `\s` is the space character;
- `\o` represents characters in cells 00-1F and 7F of row 00, plane 00, group 00 of ISO 10646.

7 Basic language elements

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

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

7.1.5.2).

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

EXAMPLE 4 – The following are equivalent

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

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

7.1 Character set

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

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

7.1.1 Digits

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

Syntax:

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

7.1.2 Letters

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

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

Syntax:

```
124 letter = 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j' | 'k' | 'l' |
            'm' | 'n' | 'o' | 'p' | 'q' | 'r' | 's' | 't' | 'u' | 'v' | 'w' | 'x' |
            'y' | 'z' .
```

7.1.3 Special characters

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

Syntax:

```

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

```

7.1.4 Underscore

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

7.1.5 Whitespace

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

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

7.1.5.1 Space character

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

7.1.5.2 Newline

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

The representation of a newline is implementation specific.

7.1.5.3 Other characters

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

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

7.1.6 Remarks

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

7.1.6.1 Embedded remark

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

Syntax:

```

142 embedded_remark = '(' { not_lparen_star | lparen_not_star | star_not_rparen |
                        embedded_remark } ')' .
126 not_lparen_star = not_paren_star | ')' .
127 not_paren_star = letter | digit | not_paren_star_special .
124 letter = 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j' | 'k' | 'l' |
            'm' | 'n' | 'o' | 'p' | 'q' | 'r' | 's' | 't' | 'u' | 'v' | 'w' | 'x' |
            'y' | 'z' .
120 digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' .
129 not_paren_star_special = not_paren_star_quote_special | '"""' .
128 not_paren_star_quote_special = '!' | '"' | '#' | '$' | '%' | '&' | '+' | ',' |
                                   '-' | '.' | '/' | ':' | ';' | '<' | '=' | '>' |
                                   '?' | '@' | '[' | '\' | ']' | '^' | '_' | '`' |
                                   '{' | '|' | '}' | '~' .
125 lparen_not_star = '(' not_star .
132 not_star = not_paren_star | '(' | ')' .
135 star_not_rparen = '*' not_rparen .
131 not_rparen = not_paren_star | '*' | '(' .

```

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

Embedded remarks may be nested.

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

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

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

7.1.6.2 Tail remark

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

Syntax:

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

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

7.2 Reserved words

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

7.2.1 Keywords

The *EXPRESS* keywords are shown in table 1.

NOTES

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

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

Table 1 – EXPRESS keywords

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

7.2.2 Reserved words which are operators

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

Table 2 – EXPRESS reserved words which are operators

AND	ANDOR	DIV	IN
LIKE	MOD	NOT	OR
XOR			

7.2.3 Built-in constants

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

Table 3 – EXPRESS reserved words which are constants

?	SELF	CONST_E	PI
FALSE	TRUE	UNKNOWN	

7.2.4 Built-in functions

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

Table 4 – EXPRESS reserved words which are function names

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

7.2.5 Built-in procedures

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

Table 5 – EXPRESS reserved words which are procedure names

INSERT	REMOVE
--------	--------

7.3 Symbols

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

Table 6 – EXPRESS symbols

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

7.4 Identifiers

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

Syntax:

```
140 simple_id = letter { letter | digit | '_' } .
124 letter = 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j' | 'k' | 'l' |
           'm' | 'n' | 'o' | 'p' | 'q' | 'r' | 's' | 't' | 'u' | 'v' | 'w' | 'x' |
           'y' | 'z' .
120 digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' .
```

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

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

7.5 Literals

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

Syntax:

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

7.5.1 Binary literal

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

Syntax:

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

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

EXAMPLE 7 – A valid binary literal

```
%0101001100
```

7.5.2 Integer literal

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

Syntax:

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

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

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

EXAMPLE 8 – Valid integer literals

```
4016
38
```

7.5.3 Real literal

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

Syntax:

```

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

```

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

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

EXAMPLES

9 – Valid real literals

```

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

```

10 – Invalid real literals

```

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

```

7.5.4 String literal

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

- first octet = ISO 10646 group in which the character is defined;
- second octet = ISO 10646 plane in which the character is defined;
- third octet = ISO 10646 row in which the character is defined;
- fourth octet = ISO 10646 cell in which the character is defined.

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

Syntax:

```

292 string_literal = simple_string_literal | encoded_string_literal .
141 simple_string_literal = \q { ( \q \q ) | not_quote | \s | \o } \q .
130 not_quote = not_paren_star_quote_special | letter | digit | '(' | ')' | '*' .
128 not_paren_star_quote_special = '!' | '"' | '#' | '$' | '%' | '&' | '+' | ',' |
                                '-' | '.' | '/' | ':' | ';' | '<' | '=' | '>' |
                                '?' | '@' | '[' | '\ ' | ']' | '^' | '_' | '`' |
                                '{' | '|' | '}' | '~' .
124 letter = 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j' | 'k' | 'l' |
            'm' | 'n' | 'o' | 'p' | 'q' | 'r' | 's' | 't' | 'u' | 'v' | 'w' | 'x' |
            'y' | 'z' .
120 digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' .
137 encoded_string_literal = ''' encoded_character { encoded_character } ''' .
122 encoded_character = octet octet octet octet .
133 octet = hex_digit hex_digit .
123 hex_digit = digit | 'a' | 'b' | 'c' | 'd' | 'e' | 'f' .

```

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

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

EXAMPLES**11 – Valid simple string literals**

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

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

12 – Invalid simple string literals

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

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

13 – Valid encoded string literals

"00000041"
 Reads A

"000000C5"
 Reads Å

14 – Invalid encoded string literals

"000041"

Octets must be supplied in groups of four

"00000041 000000C5"

Only hexadecimal characters are allowed between ""

7.5.5 Logical literal

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

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

Syntax:

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

8 Data types

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

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

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

8.1 Simple data types

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

8.1.1 Number data type

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

Syntax:

```
248 number_type = NUMBER .
```

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

size : NUMBER ;

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

8.1.2 Real data type

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

Syntax:

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

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

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

Rules and restrictions:

- a) The **precision_spec** gives the minimum number of digits of resolution that are required. This expression shall evaluate to a positive integer value.
- b) When no resolution specification is given the precision of the real number is unconstrained.

8.1.3 Integer data type

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

Syntax:

```
227 integer_type = INTEGER .
```

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

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

```
...  
END_ENTITY;
```

8.1.4 Logical data type

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

Syntax:

```
243 logical_type = LOGICAL .
```

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

8.1.5 Boolean data type

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

Syntax:

```
173 boolean_type = BOOLEAN .
```

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

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

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

8.1.6 String data type

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

Syntax:

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

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

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

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

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

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

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

Rules and restrictions:

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

EXAMPLES

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

```
string1 : STRING;
```

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

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

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

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

8.1.7 Binary data type

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

Syntax:

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

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

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

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

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

Rules and restrictions:

The width expression shall evaluate to a positive integer value.

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

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

8.2 Aggregation data types

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

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

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

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

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

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

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

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

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

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

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

8.2.1 Array data type

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

Syntax:

```

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

```

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

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

Rules and restrictions:

- a) Both expressions in the bound specification, `bound_1` and `bound_2`, shall evaluate to integer values. Neither shall evaluate to the indeterminate (?) value.
- b) `bound_1` gives the lower bound of the array. This shall be the lowest index which is valid for an array value of this data type.
- c) `bound_2` gives the upper bound of the array. This shall be the highest index which is valid for an array value of this data type.
- d) `bound_1` shall be less than or equal to `bound_2`.
- e) If the `OPTIONAL` keyword is specified, an array value of this data type may have the indeterminate (?) value at one or more index positions.
- f) If the `OPTIONAL` keyword is not specified, an array value of this data type shall not contain an indeterminate (?) value at any index position.
- g) If the `UNIQUE` keyword is specified, each element in an array value of this data type shall be different from (i.e., not instance equal to) every other element in the same array value.

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

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

```

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

```

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

8.2.2 List data type

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

Syntax:

```

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

```

Rules and restrictions:

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

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

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

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

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

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

```

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

```

8.2.3 Bag data type

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

Syntax:

```

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

```

Rules and restrictions:

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

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

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

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

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

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

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

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

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

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

8.2.4 Set data type

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

ISO 10303-11:1994(E)

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

Syntax:

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

Rules and restrictions:

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

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

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

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

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

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

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

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

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

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

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

8.2.5 Value uniqueness on aggregates

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

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

```

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

```

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

```

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

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

```

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

- a) `my_equal(i, i)` is TRUE for all i in $\mathcal{S}$ (since indeterminate (?) is not in $\mathcal{S}$, this does not require `my_equal(?, ?)` to be TRUE);
- b) `my_equal(i, j) = my_equal(j, i)` for all i, j in $\mathcal{S}$;
- c) (`my_equal(i, j) = TRUE`) AND (`my_equal(j, k) = TRUE`) implies (`my_equal(i, k) = TRUE`) for all i, j, k in $\mathcal{S}$.

8.3 Named data types

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

8.3.1 Entity data type

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

Syntax:

```
147 entity_ref = entity_id .
```

Rules and restrictions:

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

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

```
ENTITY point;  
    x, y, z : REAL;  
END_ENTITY;  
  
ENTITY line;  
    p0, p1 : point;  
END_ENTITY;
```

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

8.3.2 Defined data type

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

Syntax:

```
154 type_ref = type_id .
```

Rules and restrictions:

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

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

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

```

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

```

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

8.4 Constructed data types

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

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

8.4.1 Enumeration data type

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

Syntax:

```

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

```

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

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

Rules and restrictions:

- a) For comparison purposes, the ordering of the values of an enumeration type shall be determined by their relative position in the **enumeration_id** list: the first occurring item shall be less than the second; the second less than the third, etc.
- b) Comparison between values of different **ENUMERATION** data types is undefined.
- c) An enumeration type shall only be used as the underlying type of a defined data type.

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

```

TYPE car_can_move = ENUMERATION OF

```

```

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

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

```

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

8.4.2 Select data type

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

Syntax:

```

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

```

Rules and restrictions:

- a) Each item in the select list shall be an entity data type or a defined data type.
- b) A SELECT data type shall only be used as the underlying type of a defined data type.

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

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

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

```

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

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

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

    ENTITY nail;
        body_length : REAL;
        head_area   : REAL;

```

```

END_ENTITY;

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

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

ENTITY weld;
  composition : material_composition;
END_ENTITY;

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

```

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

8.5 Generalized data types

Syntax:

```

211 generalized_types = aggregate_type | general_aggregation_types | generic_type .

```

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

8.6 Data type usage classification

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

Table 7 – The use of data types

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

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

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

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

8.6.1 Base data types

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

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

Syntax:

171 base_type = aggregation_types | simple_types | named_types .

8.6.2 Parameter data types

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

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

Syntax:

253 parameter_type = generalized_types | named_types | simple_types .

8.6.3 Underlying data types

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

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

Syntax:

309 underlying_type = constructed_types | aggregation_types | simple_types | type_ref .