INTERNATIONAL STANDARD

ISO 10303-14

First edition
2005-10-01

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

Part 14: Description methods: The EXPRESS-X language reference manual

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

Partie 14: Méthodes descriptives: Le manuel de référence du langage EXPRESS-X

Reference number
ISO 10303-14:2005(E)

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

Contents

Tables

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 part of ISO 10303 may be the subject of patent rights. ISO shall not be held responsible for identifying any or all such patent rights.

ISO 10303-14 was prepared by Technical Committee ISO/TC 184, *Industrial automation systems and integration*, Subcommittee SC4, *Industrial data*.

This International Standard is organized as a series of parts, each published separately. The structure of this International Standard is described in ISO 10303-1.

Each part of this International Standard 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 10 series.

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

`http://www.tc184-sc4.org/SC4_Open/SC4_Work_Products_Documents/STEP_(10303)`

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

Introduction

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. The 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 is a member of the 10 series. This part of ISO 10303 specifies a language for specifying relationships between data that is governed by EXPRESS schemas, and for specifying alternative views of data that is governed by EXPRESS schemas. The language is called EXPRESS-X.

It is assumed that readers of this part of ISO 10303 are familiar with the EXPRESS data specification language defined in ISO 10303-11 and with clear text encoding specification defined in ISO 10303-21.

Industrial automation systems and integration — Product data representation and exchange — Part 14: Description methods: The EXPRESS-X language reference manual

1 Scope

This part of ISO 10303 specifies a language for specifying relationships between data that is governed by EXPRESS schemas, and for specifying alternate views of data that is governed by EXPRESS schemas. The language is called EXPRESS-X.

EXPRESS-X is a structural data mapping language. It consists of language elements that allow an unambiguous specification of a relationship between EXPRESS schemas.

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

– mapping of data governed by one EXPRESS schema to data governed by another EXPRESS schema;

– mapping of data governed by one version of an EXPRESS schema to data governed by another version of that EXPRESS schema, where the two schemas have different names;

– specification of requirements for data translators for data sharing and data exchange applications;

– specification of alternate views of data defined by an EXPRESS schema;

– an alternate notation for application protocol mapping tables;

– bi-directional mappings where mathematically possible;

– specification of constraints that may be evaluated against data produced by mapping.

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

– mapping of data defined using means other than EXPRESS;

– identification of the version of an EXPRESS schema;

– graphical representation of constructs in the EXPRESS-X language.

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

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

ISO 10303-11:2004, *Industrial automation systems and integration — Product data representation and exchange — Part 11: Description methods: The EXPRESS language reference manual.*

ISO/IEC 10646-1:2000, *Information technology — Universal Multi-Octet Coded Character Set (UCS) — Part 1: Architecture and Basic Multilingual Plane.*

3 Terms and Definitions

3.1 Terms defined in ISO 10303-1

For the purpose of this part of ISO 10303, the following terms defined in ISO 10303-1 apply:

– data;

– information.

3.2 Terms defined in ISO 10303-11

For the purpose of this part of ISO 10303, the following terms defined in ISO 10303-11 apply:

– complex entity data type;

– complex entity (data type) instance;

– constant;

– entity;

– entity data type;

– entity (data type) instance;

– instance;

– partial complex entity data type;

– partial complex entity value;

– population;

– simple entity (data type) instance;

– subtype/supertype graph;

– token;

– value.

3.3 Other definitions

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

3.3.1
binding extent
a set of binding instances constructed from instances in source entity data type extents and view extents

3.3.2
binding instance
a collection of references to entity data type instances and view data type instances associated with a view or map.

3.3.3
entity data type extent
the collection of instances of a given entity data type

3.3.4
EXPRESS-X parser
a tool capable of parsing a specification stated in the EXPRESS-X language

3.3.5
EXPRESS-X mapping engine
a tool that performs structural information mapping based on a specification stated in the EXPRESS-X language

3.3.6
map
the declaration of a relationship between data of one or more source entity data types or source view data types and data of one or more target entity data types

3.3.7
network mapping
a mapping to many target entity instances

3.3.8
qualified binding extent
a subset of a binding extent consisting of binding instances satisfying a set of selection criteria

NOTE A set of selection criteria is satisfied if each selection criterion individually is satisfied.

3.3.9
selection criterion
a logical expression, the criterion being satisfied only if the expression evaluates to the value `TRUE`

3.3.10
source data set
a collection of entity data type instances governed by an EXPRESS schema and serving as an origin of mapping

3.3.11
source extent
a view extent or entity data type extent drawn on to create a binding extent

3.3.12
target data set
a collection of entity instances produced by means of mapping

3.3.13
view
an alternative organization of the information in an EXPRESS schema

3.3.14
view data type
the representation of a view

3.3.15
view data type instance
a named unit of information established by evaluation of a view

3.3.16
view extent
an aggregate of view data type instances that contains all instances that can be constructed from the qualified binding extent

4 Fundamental principles

4.1 Overview

The following principles apply to this part of ISO 10303; the concepts described in ISO 10303-11, clause 5 also apply.

EXPRESS-X provides the specification of:

– differing views of the data governed by an EXPRESS schema, using view declarations (see 9.3) in a schema view (see 9.5);

– the mapping of data governed by one or more source EXPRESS schemas into data governed by one or more target EXPRESS schemas, using map declarations (see 9.4) in a schema map (see 9.6).

An EXPRESS-X schema may contain EXPRESS function and procedure specifications in order to support the definition of views and maps.

NOTE 1 A typographical convention used throughout this specification is to contextualize parts of a sentence to either the discussion of views or maps particularly by including the terms “(views)” or “(maps)” were appropriate in the sentence.

NOTE 2 A typographical convention used throughout this specification is that a binding instance is denoted as an ordered set of entity or view instance names separated by commas “,” and enclosed in angle brackets, “<>”. The ordering of instance names corresponds to the order of appearance of the source extent in the FROM language element of the subject view or map declaration.

EXAMPLE Given the view declaration:

```
SCHEMA_VIEW my_person_org_schema_view;
REFERENCE FROM person_and_org_schema;
VIEW person_org;
  FROM p: person; o : organization; -- provides ordering
  SELECT
    name : STRING := p.last_name;
    org : STRING := o.department_name;
END_VIEW;
END_SCHEMA_VIEW;
```

the source express schema:

```
SCHEMA person_and_org_schema;
ENTITY person;
  first_name : STRING;
  last_name : STRING;
END_ENTITY;
ENTITY organization;
  department_name : STRING;
END_ENTITY;
END_SCHEMA;
```

and the data (encoded as defined in ISO 10303-21 — see [2]):

```
#1=PERSON('James','Smith');
#2=PERSON('Fredrick','Jones');
#31=ORGANIZATION('Engineering');
#32=ORGANIZATION('Sales');
```

binding instances for this view and data may be written as below. The concept of binding instances is defined in subsequent clauses and is not necessary to understand the example. Note here, however, that the first element of each binding instance is drawn from the **person** extent and the second element is draw from the **organization** extent. This ordering corresponds to the order of appearance of **person** and **organization** in the FROM language element of the view:

```
{<#1,#31>,<#1,#32>,<#2,#31>,<#2,#32>}.
```

4.2 Fundamental principles of the execution model

4.2.1 Overview

This specification defines a language and an execution model. The execution model is composed of two phases: a binding process and an instantiation process. The evaluation of views and maps share a common binding process but differ with respect to instantiation.

4.2.2 Binding process

A binding is an environment in which values are given to variables. A binding instance is a structure that binds the variables declared in the FROM language element of a view or map declaration. The FROM language element references source entity extents and view extents. The values bound are taken from these source extents. Each binding instance is a member of the set computed as the Cartesian product of the source extents referenced. The set of binding instances thus computed is the binding extent of that view or map for the given source extents. The variable bindings of a binding instance provide an environment for the evaluation of the body of the view or map in the instantiation process, where the data referenced in the binding instance is related to structures created in the target population. Thus each

binding instance corresponds to a view data type instance (views) or target entity data type instances (maps) in the target population.

The source extents of maps and views shall be entity data type extents or view extents.

Circularity among references to source extents is prohibited.

EXAMPLE 1 The binding process applied to the view, data and schema defined in the example of 4.1 computes a binding extent of **person_org** {<#1,#31>, <#1,#32>, <#2,#31>, <#2,#32>}. This extent is depicted in tabular form below:

Binding Instance	person			organization	
	#	first_name	last_name	#	department_name
<#1,#31>	#1	'James'	'Smith'	#31	'Engineering'
<#1,#32>	#1	'James'	'Smith'	#32	'Sales'
<#2,#31>	#2	'Fredrick'	'Jones'	#31	'Engineering'
<#2,#32>	#2	'Fredrick'	'Jones'	#32	'Sales'

EXAMPLE 2 The following schema_view is invalid; it contains a cycle of references (view **a** references view **b** which references view **a**).

```
SCHEMA_VIEW invalid;
VIEW a;
  FROM some_b : b;
  attr1 : INTEGER := some_b.attr2 + 2;
END_VIEW;
VIEW b;
  FROM some_a : a;
  attr2 : INTEGER := some_a.attr1 * 3;
END_VIEW;
END_SCHEMA_VIEW;
```

4.2.3 Instantiation process

A binding is an environment in which variables are given values used during the instantiation process. Each binding instance provides a set of values to be bound to the variables. The view instantiation process is the process of evaluating the body of the view (see 9.3.2) for each binding instance in the binding extent. The order of evaluation of the binding instances is not specified.

EXAMPLE 1 The binding process applied to the schema, data, and view declaration of the example in clause 4.1 results in the binding extent of person_org: {<#1,#31>, <#1,#32>, <#2,#31>, <#2,#32>}. The view declaration and data used in that example is repeated here:

```
VIEW person_org;
  FROM p: person; o : organization; -- provides the illustrated ordering
  SELECT
    name : STRING := p.last_name;
    org : STRING := o.department_name;
END_VIEW;
```

```
#1=PERSON('James','Smith');
#2=PERSON('Fredrick','Jones');
#31=ORGANIZATION('Engineering');
#32=ORGANIZATION('Sales');
```


The binding instance <#1,#31> corresponds to the assignment of entity data type instance #1 to the variable p and #31 to variable o. Evaluation of the body of the view in this binding results in a view data type instance with name attribute 'Smith' and org attribute 'Engineering.' View data type instances may be encoded as though they were entity data type instances using the encoding specified in ISO 10303-21 [2]. The view extent for this example is:

```
#100=PERSON_ORG('Smith','Engineering');    /* <#1,#31> */
#101=PERSON_ORG('Smith','Sales');          /* <#1,#32> */
#102=PERSON_ORG('Jones','Engineering');    /* <#2,#31> */
#103=PERSON_ORG('Jones','Sales');          /* <#2,#32> */
```

EXAMPLE 2 A target EXPRESS schema and schema map with structure similar to that of the schema view used in the previous example can be defined as follows:

```
SCHEMA similar_target;
ENTITY person_org;
 name : STRING;
 org : STRING;
END_ENTITY;
END_SCHEMA;
```

```
SCHEMA_MAP similar;
REFERENCE FROM person_and_org_schema AS SOURCE;
REFERENCE FROM similar_target AS TARGET;
MAP person_org_map AS
  po : person_org;
 FROM
  p : person;
  o : organization;
 SELECT
  po.name := p.last_name;
  po.org := o.department_name;
END_MAP;
END_SCHEMA_MAP;
```

Evaluation of the data of the previous example results in the following entity data type instances governed by the schema similar_target:

```
#100=PERSON_ORG('Smith','Engineering');      /* <#1,#31> */
#101=PERSON_ORG('Smith','Sales');            /* <#1,#32> */
#102=PERSON_ORG('Jones','Engineering');      /* <#2,#31> */
#103=PERSON_ORG('Jones','Sales');            /* <#2,#32>  */
```

When the expression in the right-hand-side of an assignment of the body of the map (view) declaration consists only of attribute references (.) from a source entity, and the referenced attribute is an explicit attribute, the mapping of that target entity attribute (view attribute) is bi-directional.

EXAMPLE 3

```
MAP person_org_map AS
 po : person_org;
FROM
 p: person;
 o : organization;
SELECT
 po.name := p.last_name;          -- bi-directional
 po.org := o.department_name;     -- bi-directional
 po.industry_code := o.owning_enterprise.industry.code_num; -- bi-directional
 po.dept_number := dept_func(o.department_name); -- possibly not bi-directional
END_MAP;
```

Whether or not the attribute po.dept_number is bi-directional depends on the nature of the dept_func function.

Clause 4.2 specifies only the fundamental aspects of the execution model. Details of the binding process are described in clause 9.2. Details of the instantiation process for views are described in clause 9.3. Details of the instantiation process for maps are described in clause 9.4.

4.3 Implementation environment

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

– how references to names are resolved;

– how input and output data sets are specified;

– how maps are executed for instances that do not conform to an EXPRESS schema.

Evaluation of a view produces a view extent. Evaluation of a map may produce entity instances in the target data set. EXPRESS-X does not specify what effect modification of source data may have on view extents or target data sets after initial mapping.

5 Conformance requirements

5.1 EXPRESS-X conformance classes

5.1.1 Overview

The conformance class of an implementation of an EXPRESS-X parser or mapping engine is indicated by what portion of the language the implementation supports. Declarations are classified by language subset in Table 1 below.

Table 1: Declarations and language subset

Declaration	Subset 1	Subset 2
view declaration	•	
map declaration		•
dependent map declaration		•
constant declaration	•	•
function declaration	•	•
procedure declaration	•	•
rule declaration	•	

The implementor of an EXPRESS-X parser or mapping engine shall state any constraints that 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 for the purpose of conformance testing.

5.1.2 EXPRESS-X parser conformance classes

An implementation of an EXPRESS-X parser shall be able to parse any formal specification written in EXPRESS-X consistent with the conformance class associated with that implementation. An EXPRESS-X parser shall be said to conform to a particular level of checking (as defined in 5.1.4) if it can apply all checks required by that level (and any level below it) to a formal specification written in EXPRESS-X.

A conformance class 1 conforming EXPRESS-X parser shall parse all declarations from language subset 1 (see Table 1).

A conformance class 2 conforming EXPRESS-X parser shall parse all declarations from language subset 2 (see Table 1).

A conforming class 3 conforming EXPRESS-X parser shall parse all declarations that may appear in this part of ISO 10303.

5.1.3 EXPRESS-X mapping engine conformance classes

An implementation of an EXPRESS-X mapping engine shall be able to execute any formal specification written in EXPRESS-X consistent with the conformance class associated with that implementation. The execution of a map is relative to one or more source data sets; the specification of how these data sets are made available to the mapping engine is outside the scope of this part of ISO 10303.

A conformance class 1 conforming EXPRESS-X mapping engine shall be able to execute all declarations from language subset 1 (see Table 1).

A conformance class 2 conforming EXPRESS-X mapping engine shall be able to execute all declarations form language subset 2 (see Table 1).

A conforming class 3 conforming EXPRESS-X mapping engine shall be able to execute all declarations that may appear in this part of ISO 10303.

5.1.4 Consistency checking of EXPRESS-X parsers

5.1.4.1 Overview

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

5.1.4.2 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-X items are consistent with the scope and visibility rules defined in clause 13.

5.1.4.3 Level 2: type checking

This level consists of Level 1 checking and checking the formal specification to ensure that it is consistent with the following:

– expressions shall comply with the rules specified in clause 10 and in ISO 10303-11 clause 12;

– assignments shall comply with the rules specified in ISO 10303-11 clause 13.3.

5.1.4.4 Level 3: value checking

This level consists of Level 2 checking and checking the formal specification to ensure that it is consistent with statements of the form, 'A shall be greater than B', as specified in clause 7 to 14 of ISO 10303-11. This is limited to those places where both A and B can be evaluated from literals and/or constants.

5.1.4.5 Level 4: complete checking

This level consists of checking the formal specification to ensure that it is consistent with all stated requirements as specified in this part of ISO 10303.

6 Language specification syntax

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

The full syntax for the EXPRESS-X language is given in Annex B. 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.

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

NOTE See the bibliography for a reference describing Wirth Syntax Notation [1].

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 that 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 that cannot be expanded further. A literal is a sequence of characters enclosed in apostrophes. For an apostrophe to appear in a literal it must be written twice, that is, `''''`.

– 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.

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

– **\a** represents any character from ISO/IEC 10646-1;

– **\n** represents a newline (system dependent) (see ISO 10303-1, 7.1.5.2).

7 Basic language elements

7.1 Overview

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

The language elements of EXPRESS-X are those of the EXPRESS language defined in Clause 7 of ISO 10303-11, with the exceptions noted below.

7.2 Reserved words

The reserved words of EXPRESS-X are the keywords and the names of built-in constants, functions, and procedures. All reserved words of EXPRESS (ISO 10303-11) are reserved words of EXPRESS-X. The reserved words shall not be used as identifiers. The additional reserved words of EXPRESS-X are specified in Table 2 below.

Table 2: Additional EXPRESS-X keywords

DEPENDENT_MAP	EACH	ELSIF	END_DEPENDENT_MAP
END_MAP	END_SCHEMA_MAP	END_SCHEMA_VIEW	END_VIEW
EXTENT	IDENTIFIED_BY	INDEXING	MAP
ORDERED_BY	PARTITION	SCHEMA_MAP	SCHEMA_VIEW
SOURCE	TARGET	VIEW	

NOTE In the case that a legal EXPRESS identifier is a reserved word in EXPRESS-X, schemas using that identifier can be mapped by renaming the conflicting identifier using the `REFERENCE` language element (see 13.2).

8 Data types

8.1 Overview

The data types defined here as well as those defined in the EXPRESS language (clause 8 of ISO 10303-11) are provided as part of the language.

Every view attribute (see 9.3.2) has an associated data type.

8.2 View data type

View data types are established by view declarations (see 9.3). A view data type is assigned an identifier in the defining schema map or schema view. The view data type is referenced by this identifier.

Syntax:

```
230 view_reference = [ ( schema_map_ref | schema_view_ref ) '.' ] view_ref
    .
```

Rules and restrictions:

a) `view_reference` shall be a reference to a view visible in the current scope.

b) `view_reference` shall not refer to a dependent view (see 9.3.5).

EXAMPLE The following declaration defines a view data type named circle.

```
VIEW circle;
 FROM e : ellipse;
 WHERE (e.major_axis = e.minor_axis);
 SELECT
  radius : REAL := e.minor_axis;
  centre : point := e.centre;
END_VIEW;
```


9 Declarations

9.1 Overview

This clause defines the various declarations available in EXPRESS-X. An EXPRESS-X declaration creates a new EXPRESS-X item and associates an identifier with it. The item may be referenced elsewhere by this identifier.

EXPRESS-X provides the following declarations:

– view;

– map;

– dependent map;

– schema view;

– schema map.

In addition, an EXPRESS-X specification may contain the following declarations defined in ISO 10303-11:1994:

– constant;

– function;

– procedure;

– rule.

9.2 Binding

9.2.1 Overview

A binding extent is a set of binding instances constructed from instances in source entity data type extents and view extents. A binding extent of a population is computed as the Cartesian product of the extents referenced in the FROM language element of the view or map declaration.

A qualified binding extent is a subset of the binding extent consisting of only those binding instances for which the WHERE language element, under the binding of its variables to the values in the binding instance, returns TRUE.

NOTE Provisions defined in 9.2 apply to map declarations and view declarations. Provisions applying only to view declarations are defined in 9.3. Provisions applying only to map declarations are defined in 9.4.

9.2.2 Binding extent

The FROM language element defines the elements of binding instances in the binding extent. The FROM language element consists of one or more source parameters. Each source parameter associates an identifier with an extent.

Syntax:

```
228  view_decl = ( root_view_decl | dependent_view_decl | subtype_view_decl
     ) .
136  map_decl = MAP map_id AS target_parameter ';' { target_parameter ';' }
     ( map_subtype_of_clause subtype_binding_header map_decl_body ) | (
     binding_header map_decl_body { binding_header map_decl_body } )
     END_MAP ';' .
 47  binding_header = [ PARTITION partition_id ';' ] [ from_clause ] [
     local_decl ] [ where_clause ] [ identified_by_clause ] [
     ordered_by_clause ] .
 90  from_clause = FROM source_parameter ';' { source_parameter ';' } .
198  source_parameter = source_parameter_id ':' extent_reference .
 83  extent_reference = source_entity_reference | view_reference .
```

Rules and restrictions:

a) `source_parameter_id`s shall be unique within the scope of the map or view declaration.

The binding extent is computed as the Cartesian product of instances in the extents referenced in the `FROM` language element.

EXAMPLE A binding extent is constructed over the populations of entity data types **item** and **person**.

```
SCHEMA source_schema;  -- An EXPRESS schema
ENTITY item;
 item_number : INTEGER;
 approved_by : STRING;
END_ENTITY;
ENTITY person;
 name : STRING;
END_ENTITY;
END_SCHEMA;
```

```
SCHEMA_VIEW example;
REFERENCE FROM source_schema;
VIEW items_and_persons;
 FROM i : item; p : person;
 SELECT
  item_number : INTEGER := i.item_number;
  responsible : STRING := p.name;
END_VIEW;
END_SCHEMA_VIEW;
```

Given a population (written as ISO 10303-21 entity instances — see [2]):

```
#1=ITEM(123,'Smith');
#2=ITEM(234,'Smith');
#33=PERSON('Jones');
#44=PERSON('Smith');
```

the corresponding binding extent of items_and_person is: {<#1,#33>,<#1,#44>,<#2,#33>,<#2,#44>}.

9.2.3 Qualification of the binding extent

The `WHERE` language element defines selection criteria on binding instances in the binding extent. The `WHERE` language element, together with the source extents identified in the `FROM` language element define the qualified binding extent.

A binding is an environment in which values are given to variables. The source parameters of the FROM language element are bound to the binding instance. The domain rule expressions of the WHERE language element are evaluated with this binding. A binding instance in the binding extent is a member of the qualified binding extent if all the domain rule expressions of the WHERE language element evaluate to TRUE.

The syntax of the WHERE language element is as defined in ISO 10303-11, 9.2.2.2.

EXAMPLE The qualified binding extent consists of those pairs of **item** and **person** of the binding extent for which **person.name** is 'Smith' or 'Jones' and **item.approved_by** is 'Smith' or 'Jones'.

```
SCHEMA_VIEW example;
REFERENCE FROM source_schema;
VIEW items_and_persons;
 FROM i : item; p : person;
 WHERE (p.name = 'Smith') OR (p.name = 'Jones');
       (i.approved_by = p.name);
 SELECT
  name : STRING := p.name;
END_VIEW;
END_SCHEMA_VIEW;
```

The qualified binding extent corresponding to the data in example 1 above is: {<#1,#44>,<#2,#44>}.

9.2.4 Identification of view and target instances

A binding instance of a map or view not containing the identified_by_clause language element is identified by the values (entity data type instances) it takes from the extents referenced in the FROM language element. A binding instance of a map or view containing the identified_by_clause language element is identified by the value(s) of the expression language element in syntax rule 108. The view call (see 10.2) and map call (see 10.3) use these identification schemes.

The identified_by_clause language element defines an equivalence relation among instances in a binding extent.

Syntax:

```
107  identified_by_clause = IDENTIFIED_BY id_parameter ';' { id_parameter
     ';' } .
108  id_parameter = [ id_parameter_id ':' ] expression .
109  id_parameter_id = simple_id .
 90  from_clause = FROM source_parameter ';' { source_parameter ';' } .
198  source_parameter = source_parameter_id ':' extent_reference .
```

Rules and restrictions:

a) When used in a map declaration, an expression in the id_parameter language syntax shall not refer, through any level of indirection, to the target entity instances of the map or any of their attributes.

Two binding instances are in the same equivalence class if, for each expression of the identified_by_clause language element, evaluating the expression in the context of each of those instances produces result that are instance equal (ISO 10303-11, 12.2.2). The instantiation process produces one view instance (views) or target network (maps) for each equivalence class.

EXAMPLE This example illustrates the use of IDENTIFIED_BY.

```
SCHEMA_VIEW example;
REFERENCE FROM some_schema;
VIEW department;
 FROM e : employee;
 IDENTIFIED_BY e.department_name;
 SELECT
  name : STRING := e.department_name;
END_VIEW;
END_SCHEMA_VIEW;
```

```
SCHEMA some_schema;
ENTITY employee;
 name : STRING;
 department_name : STRING;
END_ENTITY;
END_SCHEMA;
```

```
#1=EMPLOYEE('Jones','Engineering');
#2=EMPLOYEE('Smith','Sales');
#3=EMPLOYEE('Doe','Engineering');
```

Given the view and population above, there are two equivalence classes: {<#1>,<#3>} and {<#2>}, corresponding to binding instances with e.department_name = 'Engineering' and e.department_name = 'Sales' respectively.

9.2.5 Equivalence classes and the instantiation process

View attributes (see 9.3.2) and target entity attributes (see 9.4.2) represent properties of the corresponding view (view declaration) and target network entities (map declaration). These attributes are provided values by evaluation of the corresponding `expression` (syntax rule 224). The expressions are evaluated in the context of a binding instance in the qualified binding extent.

Syntax:

```
224  view_attribute_decl = view_attribute_id ':' [ OPTIONAL ] [
     source_schema_ref '.' ] base_type ':=' expression ';' .
134  map_attribute_declaration = [ target_parameter_ref [ index_qualifier ]
     [ group_qualifier ] '.' ] attribute_ref [ index_qualifier ] ':='
     expression ';' .
```

If an equivalence class defined by an `identified_by_clause` language element contains more than one qualified binding instance, then the value of the `expression` is computed as follows:

– if there are binding instances for which `expression` (syntax rule 224) evaluates to a non-indeterminate value, and if all such non-indeterminate values are the same (instance equal) or if there is only one such value, that value is assigned to the attribute.

– If for two or more binding instances, the evaluation of the `expression` of the attribute produces non-indeterminate, unequal values (instance equal), or if all evaluations produce indeterminate values, an indeterminate value is assigned to the attribute.

EXAMPLE This example illustrates the assignment of values where an equivalence class contain more than one qualified binding instance. The map declaration is described in clause 9.4.

```
(* source schema *)           (* target schema *)
SCHEMA src;                  SCHEMA tar;
ENTITY employee;             ENTITY department;
 name : STRING;               employee : STRING;
 manager : STRING;            manager : STRING;
 dept : STRING;               dept_name : STRING;
END_ENTITY;                  END_ENTITY;
END_SCHEMA;                 END_SCHEMA;
```

```
(* mapping schema *)
SCHEMA_MAP example;
REFERENCE FROM src AS SOURCE;
REFERENCE FROM tar AS TARGET;
MAP department_map AS d : department;
 FROM e : src.employee;
 IDENTIFIED_BY e.dept;
 SELECT
  d.employee := e.name;
  d.manager := e.manager;
  d.dept_name := e.dept;
END_MAP;
END_SCHEMA_MAP;
```

```
#1=EMPLOYEE('Smith','Jones','Marketing');
#2=EMPLOYEE('Doe','Jones','Marketing');
```

Given the data above, the target data set contains one entity instance, #1=DEPARTMENT($,'Jones','Marketing'). The attribute **department.dept_name** is indeterminate because the expression for this attribute evaluates to two different values ('Smith' and 'Doe').

9.2.6 Ordering of view and target instances

The ORDERED_BY language element defines an ordering on the binding instances of a qualified binding extent. Partial binding calls (see 10.4) on the corresponding view or map partition evaluate to aggregates ordered according to the ORDERED_BY language element, if present.

The expression in an ORDERED_BY language element shall evaluate to values of the same order-comparable type for all bindings in the partition. The order-comparable types are NUMBER, BINARY, STRING, and ENUMERATION and specializations of these. A resulting ordering on the binding extent shall be such that the expression ($e < f$) shall not evaluate to FALSE, where e and f are respectively the values produced by evaluation of the expression (see syntax rule 148) for any two successive elements of the binding extent, and < is the EXPRESS value comparison operator (ISO 10303-11, 12.2.1). If additional expression language elements are specified, then for each subsequent expression, the process describe is applied to the result of the previous expression.

Syntax:

```
 47  binding_header = [ PARTITION partition_id ';' ] [ from_clause ] [
     local_decl ] [ where_clause ] [ identified_by_clause ] [
     ordered_by_clause ] .
148  ordered_by_clause = ORDERED_BY expression { ',' expression } ';' .
```

Rules and restrictions:

a) The expression of syntax rule 148 shall not evaluate to an indeterminate value for any bindings in the partition.

b) Subtype view and map partitions shall not specify an `ORDERED_BY` language element. Subtype views and map partitions shall inherit ordering from parent partitions defining `ORDERED_BY`, if present in a parent.

c) A collection of view partitions related through type hierarchy shall contain no more than one `ORDERED_BY` language element.

d) A collection of map partitions related through the `map_subtype_of_clause` (see syntax rule 141) shall contain no more than one `ORDERED_BY` language element.

9.3 View declaration

9.3.1 Overview

A view declaration creates a view data type and declares an identifier to refer to it.

EXAMPLE The following view defines a view data type **arm_person_role_in_organization**.

```
VIEW arm_person_role_in_organization;
FROM
 pao : person_and_organization;
 ccdpaoa : cc_design_person_and_organization_assignment;
WHERE ccdpaoa.assigned_person_and_organization :=: pao;
SELECT
 person : person := pao.the_person;
 org : organization := pao.the_organization;
 role : label := ccdpaoa.role.name;
END_VIEW;
```

Syntax:

```
228  view_decl = ( root_view_decl | dependent_view_decl | subtype_view_decl
     ) .
177  root_view_decl = VIEW view_id [ supertype_constraint ] ';'
     binding_header SELECT view_attr_decl_stmt_list { binding_header SELECT
     view_attr_decl_stmt_list } END_VIEW ';' .
 47  binding_header = [ PARTITION partition_id ';' ] [ from_clause ] [
     local_decl ] [ where_clause ] [ identified_by_clause ] [
     ordered_by_clause ] .
 90  from_clause = FROM source_parameter ';' { source_parameter ';' } .
198  source_parameter = source_parameter_id ':' extent_reference .
 83  extent_reference = source_entity_reference | view_reference .
```

9.3.2 View attributes

An attribute of a view data type represents a property of the view. The value of the attribute of a view instance is computed as the evaluation of the `expression` of syntax rule 224.

The name of a view attribute, `view_attribute_id` of syntax rule 224, represents the role played by its associated value in the context of the view in which it appears.

Syntax:

```
226  view_attr_decl_stmt_list = { view_attribute_decl } .
224  view_attribute_decl = view_attribute_id ':' [ OPTIONAL ] [
     source_schema_ref '.' ] base_type ':=' expression ';' .
```

Rules and restrictions:

a) The value resulting from evaluation of the `expression` of syntax rule 224 shall be assignment compatible with the `base_type` of the view attribute.

b) Each `view_attribute_id` declared in the view declaration shall be unique within that declaration.

c) `OPTIONAL` indicates that the value of the attribute may be indeterminate. Use of `OPTIONAL` has no effect on the execution model.

9.3.3 View partitions

A view partition is a subset of a view extent. A view extent is the union of its partitions. A view declaration consists of one or more partition declarations, each partition declaration having its own `FROM` and `WHERE` language elements.

EXAMPLE In ISO 10303-201, the application object **ORGANIZATION** may be mapped to from a **PERSON**, an **ORGANIZATION**, or a **PERSON_AND_ORGANIZATION** entity. A view from this schema to the arm_organization view is specified as follows:

```
VIEW arm_organization;
PARTITION a_single_person;
  FROM p : person;
    ...
PARTITION a_single_organization;
  FROM o: organization;
    ...
PARTITION a_person_in_an_organization;
  FROM po: person_and_organization;
    ...
END_VIEW;
```

Syntax:

```
228  view_decl = ( root_view_decl | dependent_view_decl | subtype_view_decl
     ) .
177  root_view_decl = VIEW view_id [ supertype_constraint ] ';'
     binding_header SELECT view_attr_decl_stmt_list { binding_header SELECT
     view_attr_decl_stmt_list } END_VIEW ';' .
 67  dependent_view_decl = VIEW view_id ':' base_type ';' binding_header
     RETURN expression { binding_header RETURN expression } END_VIEW ';' .
206  subtype_view_decl = VIEW view_id subtype_declaration ';'
     subtype_binding_header SELECT view_attr_decl_stmt_list {
     subtype_binding_header SELECT view_attr_decl_stmt_list } END_VIEW ';'
     .
203  subtype_binding_header = [ PARTITION partition_id ';' ] where_clause .
 47  binding_header = [ PARTITION partition_id ';' ] [ from_clause ] [
     local_decl ] [ where_clause ] [ identified_by_clause ] [
     ordered_by_clause ] .
```


Rules and restrictions:

a) If more than one partition exists in the view declaration, a `partition_id` shall be provided for each partition.

b) Each `partition_id` within a view declaration shall be unique.

c) All partitions of a view declaration shall define the same attributes (including names and types).

d) The attributes of a view declaration shall appear in the same order in each of its partitions.

9.3.4 Constant partitions

A partition that omits the `FROM`, `WHERE`, and `IDENTIFIED_BY` language elements is called a constant partition. Such a partition represents a single view instance with no correspondence to the source data.

EXAMPLE This example illustrates the use of constant partitions.

```
VIEW person;
PARTITION mary;
  SELECT
    name : STRING := 'Mary';
    age : INTEGER := 22;
PARTITION john;
   SELECT
    name : STRING := 'John';
    age : INTEGER := 23;
END_VIEW;
```

9.3.5 Dependent views

A dependent view is a view that does not define attributes. Partitions of the dependent view specify `RETURN expression` (see syntax rule 67). Evaluation of `expression` shall not produce a value of type `AGGREGATE`. The value computed shall be type compatible with `base_type` (see syntax rule 67).

Syntax:

```
228  view_decl = ( root_view_decl | dependent_view_decl | subtype_view_decl
     ) .
 67  dependent_view_decl = VIEW view_id ':' base_type ';' binding_header
     RETURN expression { binding_header RETURN expression } END_VIEW ';' .
```

If an equivalence class defined by an `IDENTIFIED_BY` language element of a dependent view contains more than one qualified binding instance, then the value returned is computed as follows:

– If for each such binding, the `RETURN` expression (syntax rule 67) produces an equal value, that value is returned;

– If for two or more bindings, the `RETURN` expression (syntax rule 67) produces unequal values, an indeterminate value is returned.

EXAMPLE 1 This example defines a subtype of type car whose instances have the value 'red' in their **colour** attribute.

```
VIEW red_car : car;
 FROM rc : car;
 WHERE rc.colour ='red';
 RETURN rc;
END_VIEW;
```

EXAMPLE 2 This example defines an extent whose members are strings. The strings come from two sources.

```
VIEW owner_name : STRING;
PARTITION one;
  FROM po : person;
   RETURN po.name;
PARTITION two;
  FROM or : organization;
   RETURN org.name;
END_VIEW;
```

9.3.6 Specifying subtype views

EXPRESS-X allows for the specification of views as subtypes of other views, where a subtype view data type is a specialization of its supertype. This establishes an inheritance relationship between the view data types in which the subtype inherits the attributes and selection criteria of its supertype. A view is a subtype view if it contains a `SUBTYPE` language element. The extent of a subtype view data type is a subset of the extent of its supertype as defined by the selection criteria defined by the `WHERE` language element in the subtype.

A subtype view inherits attributes from its supertype view(s). Inheritance of attributes shall adhere to the rules and restrictions of attribute inheritance defined in ISO 10303-1, 9.2.3.3.

A subtype view declaration may redefine attributes found in one of its supertypes. The redefinition of attributes shall adhere to the rules and restrictions of attribute redefinition defined in ISO 10303-11, 9.2.3.4.

During the evaluation of a view, a view instance shall be created if the selection criteria of the most general supertype is satisfied. The view instance shall have the type corresponding to a subtype view if all of the selection criteria in that subtype view in addition to all of its supertype views evaluate to `TRUE`.

Syntax:

```
228 view_decl = ( root_view_decl | dependent_view_decl | subtype_view_decl
    ) .
206 subtype_view_decl = VIEW view_id subtype_declaration ';'
    subtype_binding_header SELECT view_attr_decl_stmt_list {
    subtype_binding_header SELECT view_attr_decl_stmt_list } END_VIEW ';'
    .
205 subtype_declaration = SUBTYPE OF '(' view_ref { ',' view_ref } ')' .
203 subtype_binding_header = [ PARTITION partition_id ';' ] where_clause .
```


Rules and restrictions:

a) Exactly one supertype view of a subtype view shall define a FROM language element

b) The set of partitions of a subtype view shall be a subset of the set of partitions of its supertype view.

EXAMPLE 1 The following view illustrates subtyping. The view **male** defines an additional membership requirement (gender = 'M') for view instances of the subtype.

```
VIEW person;
 FROM e:employee;
END_VIEW;

VIEW male SUBTYPE OF (person);
 WHERE e.gender = 'M';
...
END_VIEW;
```

EXAMPLE 2 This example illustrates the use of partitions and subtype views.

```
VIEW j;
PARTITION first;
 FROM s:three, t:four
 WHERE cond6;
...
PARTITION second;
 FROM r:four, q:five
 WHERE cond7;
...
END_VIEW;

VIEW k SUBTYPE OF (j);
 PARTITION second;
 WHERE cond9;
...
END_VIEW;
```

Any subtype view for which 'k' is a supertype can only include partition 'second.'

9.3.7 Supertype constraints

A view declaration may define supertype constraints (see ISO 10303-11, 9.2.4). Whether or not a supertype constraint is satisfied has no effect on the execution model nor on the contents of view extents.

Syntax:

```
228  view_decl = ( root_view_decl | dependent_view_decl | subtype_view_decl
     ) .
177  root_view_decl = VIEW view_id [ supertype_constraint ] ';'
     binding_header SELECT view_attr_decl_stmt_list { binding_header SELECT
     view_attr_decl_stmt_list } END_VIEW ';' .
207  supertype_constraint = abstract_supertype_declaration | supertype_rule
     .
 33  abstract_supertype_declaration = ABSTRACT SUPERTYPE [
     subtype_constraint ] .
205  subtype_declaration = SUBTYPE OF '(' view_ref { ',' view_ref } ')' .
204  subtype_constraint = OF '(' supertype_expression ')' .
208  supertype_expression = supertype_factor { ANDOR supertype_factor } .
209  supertype_factor = supertype_term { AND supertype_term } .
211  supertype_term = view_ref | one_of | '(' supertype_expression ')' .
147  one_of = ONEOF '(' supertype_expression { ',' supertype_expression }
     ')' .
```


EXAMPLE

```
VIEW a ABSTRACT SUPERTYPE OF ONEOF(b ANDOR c, d);
 ...
END_VIEW;
```

An instance of **a** is valid if it has at least two types (**a** and something else) because of the ABSTRACT keyword, and one of the other types is either **d** or some combination of **b** and **c** because of the ONEOF keyword.

9.4 Map declaration

9.4.1 Overview

The map declaration supports the specification of correspondence between entity data type definitions of two or more EXPRESS schemas. The declaration supports the mapping from many source entity data type definitions to many target entity data type definitions.

Syntax:

```
136  map_decl = MAP map_id AS target_parameter ';' { target_parameter ';' }
     ( map_subtype_of_clause subtype_binding_header map_decl_body ) | (
     binding_header map_decl_body { binding_header map_decl_body } )
     END_MAP ';' .
 47  binding_header = [ PARTITION partition_id ';' ] [ from_clause ] [
     local_decl ] [ where_clause ] [ identified_by_clause ] [
     ordered_by_clause ] .
203  subtype_binding_header = [ PARTITION partition_id ';' ] where_clause .
 90  from_clause = FROM source_parameter ';' { source_parameter ';' } .
198  source_parameter = source_parameter_id ':' extent_reference .
 83  extent_reference = source_entity_reference | view_reference .
137  map_decl_body = ( entity_instantiation_loop {
     entity_instantiation_loop } ) | map_project_clause | ( RETURN expres-
     sion ';' ) .
214  target_parameter = target_parameter_id { ',' target_parameter_id } ':'
     [ AGGREGATE [ bound_spec ] OF ] target_entity_reference .
213  target_entity_reference = entity_reference { '&' entity_reference } .
```

Rules and restrictions:

a) The `map_id` of syntax rule 136 names a map declaration.

EXAMPLE In the example below, a **pump** in the source data set is mapped to a **product** and **product_related_product_category**.

```
SCHEMA source_schema;
ENTITY pump;
 id, name : STRING;
END_ENTITY;

END_SCHEMA;

SCHEMA target_schema;
ENTITY product;
 id, name : STRING;
END_ENTITY;

ENTITY product_related_product_category;
 name : STRING;
 products : SET OF product;
END_ENTITY;

END_SCHEMA;
```



```
SCHEMA_MAP pump_mapping;
REFERENCE FROM source_schema AS SOURCE;
REFERENCE FROM target_schema AS TARGET;
MAP network_for_pump AS
  pr : product;
  prpc : product_related_product_category;
FROM p : pump;
 SELECT
  pr.id := p.id;
  pr.name := p.name;
  prpc.name := 'pump';
  prpc.products := [ pr ];
END_MAP;
END_SCHEMA_MAP;
```

Note that, in this example, for each instance of type `product` created there is exactly one instance of `product_related_product_category` created.

The initial values of the attributes of the newly created instance(s) are indeterminate. If the attribute is not assigned in the body of the map the value remains indeterminate.

9.4.2 Evaluation of the map body

Syntax:

```
136 map_decl = MAP map_id AS target_parameter ';' { target_parameter ';' }
    ( map_subtype_of_clause subtype_binding header map_decl_body ) | (
    binding_header map_decl_body { binding_header map_decl_body } )
    END_MAP ';' .
137 map_decl_body = ( entity_instantiation_loop {
    entity_instantiation_loop } ) | map_project_clause | ( RETURN expres-
    sion ';' ) .
139 map_project_clause = SELECT { map_attribute_declaration } .
134 map_attribute_declaration = [ target_parameter_ref [ index_qualifier ]
    [ group_qualifier ] '.' ] attribute_ref [ index_qualifier ] ':='
    expression ';' .
```

A `map_decl_body` specifying `map_attribute_declaration` syntax elements shall assign values to the attributes of the target entity instances. The `expression` shall produce a value that is assignment compatible with the target entity attribute (see ISO 10303-11, 13.3).

A `map_decl_body` specifying `RETURN` shall evaluate the expression which is specified after the `RETURN` keyword. Evaluation shall result in the instantiation of target entity instances that are type compatible with the entity data types defined by the target parameters. Entity instances for all target parameters shall be instantiated.

9.4.3 Iteration under a single binding instance

9.4.3.1 Overview

Evaluation of a map may produce aggregates of target entity data type instances. The initial value of the aggregate is indeterminate.

The instantiation_loop_control and repeat_control provide the following forms of iteration:

– iteration over the collection of instances in an EXPRESS aggregate;

– iteration incrementing a numeric variable.

Syntax:

```
 77  entity_instantiation_loop =  FOR instantiation_loop_control ';'
     map_project_clause .
139  map_project_clause = SELECT { map_attribute_declaration } .
119  instantiation_loop_control = instantiation_foreach_control |
     repeat_control .
118  instantiation_foreach_control = EACH variable_id IN expression { AND
     variable_id IN expression } [ INDEXING variable_id ] .
171  repeat_control = [ increment_control ] [ while_control ] [
     until_control ] .
```

Rules and restrictions:

a) The map_project_clause language element (see 139) establishes a local scope in which all the loop variables variable_id (see syntax rule 118) are implicitly declared.

b) The type of the variable implicitly declared by variable_id before IN (see syntax rule 118) is the type of the expression.

c) The type of the variable implicitly declared by variable_id after INDEXING (see syntax rule 118) is of type INTEGER.

The variable_id after INDEXING (see syntax rule 118) is initialized to one at the beginning of the first iteration cycle and incremented by one at the beginning of each subsequent cycle.

9.4.3.2 Control by numeric increment

The repeat_control syntax element allows for the iteration under a single binding instance by means of the EXPRESS language repeat_control (see ISO 10303-11, 13.9).

EXAMPLE This example illustrates the use of the EXPRESS `repeat_control` in EXPRESS-X target instantiation. A collection of target **child** entity instances are created for each source **parent** entity. The number created is specified by the **parent** entity attribute **number_of_children**.

```
SCHEMA src;                          SCHEMA tar;
ENTITY parent;                       ENTITY parent;
number_of_children : INTEGER;        END_ENTITY;
END_ENTITY;                          ENTITY child;
END_SCHEMA;                             parent : parent;
                                     END_ENTITY;
                                     END_SCHEMA;


SCHEMA_MAP example;
  REFERENCE FROM src AS SOURCE;
  REFERENCE FROM tar AS TARGET;

MAP parent_map AS tp : tar.parent;
 FROM sp : src.parent;
 SELECT
END_MAP;

MAP children_map AS c : AGGREGATE OF child;
 FROM p : src.parent;
 FOR i := 1 TO p.number_of_children;
 SELECT
  c[i].parent := tp(p);
END_MAP;
END_SCHEMA_MAP;
```

9.4.3.3 Control by iteration over an aggregate

Under the `instantiation_foreach_control` syntax element, at each iteration step, the next element of the specified aggregate is bound to a variable and optionally the index position of that element is bound to an iterator variable. The scope of these variable bindings includes the `map_project_clause`.

EXAMPLE In the following example, all item versions of one item are grouped together in the source data set. In the target set, each item version is an instance.

```
SCHEMA tar;
ENTITY item_version;
  item_id    : INTEGER;
  version_id : INTEGER;
END_ENTITY;
END_SCHEMA;

SCHEMA src;
ENTITY item_with_versions;
  id             : INTEGER;
  id_of_versions : LIST OF INTEGER;
END_ENTITY;
END_SCHEMA;


SCHEMA_MAP mapping_schema;
REFERENCE FROM src AS SOURCE;
REFERENCE FROM tar AS TARGET;
```



```
MAP item_version_map AS
 iv : AGGREGATE OF item_version;
FROM
 iwv : item_with_versions;
FOR EACH version_iterator IN iwv.id_of_versions INDEXING i;
SELECT
  iv[i].item_id    := iwv.id;
  iv[i].version_id := version_iterator;
END_MAP;

END_SCHEMA_MAP;
```

For example, the following target instances are built from the source instance below.
Source instance set:

```
  #1 = ITEM_WITH_VERSIONS(1,(10,11,12));
```

Target instance set:

```
  #1 = ITEM_VERSION(1,10);
  #2 = ITEM_VERSION(1,11);
  #3 = ITEM_VERSION(1,12);
```

The `instantiation_foreach_control` syntax element may specify many expressions using the optional `AND` syntax (see syntax rule `118`). Iteration continues while at least one source aggregate is not exhausted. An indeterminate value is assigned to the `variable_id` of exhausted aggregates.

9.4.4 Map partitions

The instances of an entity data type may each relate differently to source data. Multiple map partitions may be used to specify these differing relations.

If multiple target entities are listed in the header of the map declaration, different subsets of those entities may be created by each partition.

Syntax:

```
136  map_decl = MAP map_id AS target_parameter ';' { target_parameter ';' }
     ( map_subtype_of_clause subtype_binding_header map_decl_body ) | (
     binding_header map_decl_body { binding_header map_decl_body } )
     END_MAP ';' .
 47  binding_header = [ PARTITION partition_id ';' ] [ from_clause ] [
     local_decl ] [ where_clause ] [ identified_by_clause ] [
     ordered_by_clause ] .
```

Rules and restrictions:

a) If more than one partition exists in the map declaration, a `partition_id` shall be provided for each partition.

b) Each `partition_id` within a map declaration shall be unique.

c) For every target entity data type referenced in the map header, at least one of the partitions of the map declaration shall create instances for it.

EXAMPLE This example illustrates how various source entity data types may be mapped into a single target entity data type using a map declaration containing partitions.

```
(*  source schema *)              (* target schema *)
SCHEMA src;                       SCHEMA tar;
ENTITY student;                   ENTITY person;
     name : STRING;                 name : STRING;
END_ENTITY;                       END_ENTITY;
ENTITY employee;                  END_SCHEMA;
  name : STRING;
END_ENTITY;
END_SCHEMA;


(* mapping schema *)
SCHEMA_MAP example;
REFERENCE FROM src AS SOURCE;
REFERENCE FROM tar AS TARGET;


MAP student_employee_to_person AS p : tar.person;
PARTITION student;
FROM s : src.student;
SELECT
  p.name  := s.name;
PARTITION employee;
FROM e : src.employee;
SELECT
  p.name := e.name;
END_MAP;
END_SCHEMA_MAP;
```

9.4.5 Mapping to a type and its subtypes

Map declarations can be organized into a type/subtype hierarchy. Subtype map declarations may extend the collection of entity instances created by its supertype map, specialize those instances created and require additional selection criteria beyond those specified in the supertype map. The specification of a target attribute assignment declared in a supertype map is inherited by its subtype maps. Through these means, the pattern of inheritance present in the target schema can be duplicated in the map declarations.

Syntax:

```
136 map_decl = MAP map_id AS target_parameter ';' { target_parameter ';' }
    ( map_subtype_of_clause subtype_binding_header map_decl_body ) | (
    binding_header map_decl_body { binding_header map_decl_body } )
    END_MAP ';' .
141 map_subtype_of_clause = SUBTYPE OF '(' map_reference ')' ';' .
```

The following rules shall apply for inheritance among map declarations:

– A subtype map declaration shall execute if its `WHERE` rule and the `WHERE` rules of all its supertype maps evaluates to `TRUE`.

– A subtype map declaration inherits all target parameters from its supertype map declarations.

– A subtype map declaration may redeclare the type of any target parameter of its supertype maps. The type of the redeclared target parameter shall be a specialization of every type of the target parameter declared in its supertype maps. The rules of specialization are those defined in ISO 10303-11, 9.2.6.

– A subtype map declaration may add new target parameters by declaring an additional `target_parameter` with an `target_parameter_id` different from the ones in its supertype map declarations.

– If a supertype map declares a target parameter of type `SELECT` (see ISO 10303-11, 8.4.2), `GENERIC` (see ISO 10303-11, 9.5.3.2), or of an entity data type that is declared `ABSTRACT` (see ISO 10303-11, 9.2.4.1) and no subtype map declaration that redefines this target parameter is executed, then no instance shall be created for the target parameter.

Whether a subtype map extends the collection of entity instances created by its supertype map or specializes those instance created depends on whether the subtype map references `target_parameter_id` syntax elements declared in the supertype map or whether it declare additional `target_parameter_id` syntax elements. A subtype map may introduce a `target_parameter_id` that is not defined in any of the supertype maps. In this case, a new target entity of the type defined by the target parameter is created.

A subtype map may reference for assignment a target attribute referenced for assignment in one of its supertypes (possibly through several levels of inheritance). In this case, the target attribute is assigned the value corresponding to that of the most specialized map for which the selection criteria and selection criteria of its supertypes is satisfied.

A subtype map shall have exactly one direct supertype map.

EXAMPLE 1 This example illustrates assignment to attributes declared in supertypes and subtypes through supertype and subtype maps. Source entities are of one type, **s_project**. Target entities are of type **t_project** and perhaps one of its subtypes, **in_house_project** and **external_project**. The **target_parameter_id**, **tp**, used in the supertype map (**project_map**) is used again in its subtype maps (**in_house_map**, **ext_map**) signifying that the corresponding target entity is specialized in the subtype maps.

```
SCHEMA source_schema;
ENTITY s_project;
  name : STRING;
  project_type : STRING;
  cost : INTEGER;
  price : INTEGER;
  vendor : STRING;
END_ENTITY;
END_SCHEMA;
```



```
SCHEMA target_schema;
ENTITY t_project
 SUPERTYPE OF (ONEOF (in_house_project, external_project));
  name : STRING;
  cost : INTEGER;
  management : STRING;
END_ENTITY;

ENTITY in_house_project
 SUBTYPE OF (t_project);
END_ENTITY;

ENTITY external_project
 SUBTYPE OF (t_project);
  price : INTEGER;
END_ENTITY;
END_SCHEMA;


SCHEMA_MAP example;
REFERENCE FROM source_schema AS SOURCE;
REFERENCE FROM target_schema AS TARGET;
MAP project_map AS tp : target_schema.t_project;
 FROM p : source_schema.s_project;
 SELECT
     tp.name := p.name;
     tp.cost := p.cost;
END_MAP;

MAP in_house_map AS tp : target_schema.in_house_project;
 SUBTYPE OF (project_map);
 WHERE (p.project_type = 'in house');
 SELECT
  tp.management := IF (p.cost < 50000) THEN 'small accts'
                   ELSE 'large accts' END_IF;
END_MAP;

MAP ext_map AS tp : target_schema.external_project;
 SUBTYPE OF (project_map);
 WHERE (p.project_type = 'external');
 SELECT
  tp.price := p.price;
  tp.management := p.vendor;
END_MAP;
END_SCHEMA_MAP;
```

A supertype map may define entity instantiation loops. A subtype map of such a supertype map shall inherit these instantiation loops. The body of the instantiation loop may be redefined. The correspondence between supertype map bodies and subtype map bodies where instantiation loops are used is made through use of identical index identifiers; the map body of the subtype map inheriting a loop shall reference the identical index identifier as defined in its supertype map.

EXAMPLE 2 This example illustrates the inheritance of an entity instantiation loop.

```
SCHEMA source_schema;
ENTITY part;
  name : STRING;
  no_of_versions : INTEGER;
  is_assembly : BOOLEAN;
END_ENTITY;
END_SCHEMA;

SCHEMA target_schema;
ENTITY product;
  name : STRING;
END_ENTITY;

ENTITY product_version;
  version_id : INTEGER;
  of_product : product;
END_ENTITY;

ENTITY product_definition;
  name : STRING;
  of_version : product_version;
END_ENTITY;

END_SCHEMA;

SCHEMA_MAP example;
REFERENCE FROM source_schema AS SOURCE;
REFERENCE FROM target_schema AS TARGET;

MAP super_map AS
  pvw   : AGGREGATE OF product_definition;
  pver  : AGGREGATE OF product_version;
  pro    : product;
 FROM prt : part;
 FOR i := 1 TO prt.no_of_versions;
 SELECT
  pver[i].version_id := i;
  pver[i].of_product := pro;
  pvw[i].of_version := pver[i];
  pvw[i].name := 'view of part ' + prt.name;
SELECT
  pro.name := 'part ' + prt.name;
END_MAP;

MAP sub_map AS
  pro    : product;
 SUBTYPE_OF (super_map);
 WHERE
  prt.is_assembly = TRUE;
 SELECT
  pvw[i].name := 'view of assembly ' + prt.name;
  pro.name := 'assembly ' + prt.name;
END_MAP;
END_SCHEMA_MAP;
```

9.4.6 Explicit declaration of complex entity data types

Complex entity data types (see ISO 10303-11, 3.2.1) may be explicitly declared in the map header. A complex entity data type is denoted by syntax that lists the partial complex entity data types that are combined to form it, separated by '&'.

The partial complex entity data types may be listed in any order.

Syntax:

```
214  target_parameter = target_parameter_id { ',' target_parameter_id } ':'
     [ AGGREGATE [ bound_spec ] OF ] target_entity_reference .
213  target_entity_reference = entity_reference { '&' entity_reference } .
 78  entity_reference = [ ( source_schema_ref | target_schema_ref |
     schema_ref ) '.' ] entity_ref .
```

Rules and restrictions:

a) Each `entity_ref` shall be a reference to an entity data type definition which is visible in the current scope.

b) The referenced complex entity data type shall describe a valid domain within a target schema (see ISO 10303-11, annex B).

c) A given `entity_reference` shall occur at most once within a `target_entity_reference`.

d) For each `entity_reference` declared in the `target_entity_reference`, none of its supertypes shall be declared.

EXAMPLE This example illustrates the use of the '&' syntax to define target complex entity data types.

```
SCHEMA source_schema;

ENTITY pump;
 id, name : STRING;
END_ENTITY;

END_SCHEMA;

SCHEMA target_schema;

ENTITY item
ABSTRACT SUPERTYPE OF (ONEOF (product, kitchen_appliance));
END_ENTITY;

ENTITY product SUBTYPE OF (item);
 id, name :STRING;
END_ENTITY;

ENTITY kitchen_appliance SUBYPE OF (item);
END_ENTITY;

ENTITY product_related_product_category;
 name : STRING;
 products : SET OF product;
END_ENTITY;

END_SCHEMA;

SCHEMA_MAP example;

REFERENCE FROM source_schema AS SOURCE;
REFERENCE FROM target_schema AS TARGET;

MAP network_for_pump AS
 pr : product & kitchen_appliance;
 prpc : product_related_product_category;
FROM p : pump;
 SELECT
  pr.id := p.id;
  pr.name := p.name;
  prpc.name := 'pump';
  prpc.products := [ pr ];
END_MAP;
```



```
END_SCHEMA_MAP;
```

9.4.7 Dependent map

A dependent map is a map that shall only execute by means of a map call (see 10.3). A dependent map shall have simple types and/or entity data types as source parameters.

Syntax:

```
 66  dependent_map_decl = DEPENDENT_MAP map_id AS target_parameter {
     target_parameter } [ map_subtype_of_clause ] dep_map_partition {
     dep_map_partition } END_DEPENDENT_MAP ';' .
214  target_parameter = target_parameter_id { ',' target_parameter_id } ':'
     [ AGGREGATE [ bound_spec ] OF ] target_entity_reference .
 71  dep_map_partition = [ PARTITION partition_id ':' ] dep_map_decl_body .
 70  dep_map_decl_body = dep_binding_decl map_project_clause .
 68  dep_binding_decl = dep_from_clause [ where_clause ] [
     ordered_by_clause ] .
 69  dep_from_clause = FROM dep_source_parameter ';' { dep_source_parameter
     ';' } .
 72  dep_source_parameter = source_parameter_id { ',' source_parameter_id }
     ':' ( simple_types | type_reference ) .
```

Rules and restrictions:

a) If more than one partition exists, a `partition_id` shall be provided for each partition.

b) For every target entity data type referenced in the dependent map header, at least one of the partitions of the map declaration shall create instances for it.

c) `Partition_ids` shall be unique within the scope of the dependent map declaration.

d) A dependent map declaration containing the `map_subtype_of_clause` shall contain exactly one partition

EXAMPLE

This example illustrates the use of a dependent map to instantiate target **organization** instances having unique **id** attributes. The call to the dependent map ensures **organization** instances in the target population have unique **id** attributes, since map calls with identical arguments return the same target entity instance. See 10.3.

```
SCHEMA source;
ENTITY named_organization;
  name : STRING;
END_ENTITY;
ENTITY id_organization;
  id : STRING;
END_ENTITY;
END_SCHEMA;

SCHEMA target_schema;
ENTITY organization;
 id : STRING;
UNIQUE
 ur1: id;
END_ENTITY;
END_SCHEMA;

SCHEMA_MAP example;

REFERENCE FROM source_schema AS SOURCE;
REFERENCE FROM target_schema AS TARGET;
```



```
MAP unique_orgs_map AS org : organization;
 PARTITION a_org;
 FROM a : named_organization;
   RETURN org_map(a.name);
 PARTITION b_org;
 FROM b : id_organization;
   RETURN org_map(b.id);
 END_MAP;
DEPENDENT_MAP org_map AS org : organization;
 FROM id : STRING;
 SELECT
   org.id := id;
END_DEPENDENT_MAP;
END_SCHEMA_MAP;
```

9.5 Schema view declaration

A schema view declaration defines a common scope for a collection of related declarations. A schema view may contain any declarations from language subset 1 (see Table 1).

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

Declarations in one schema view or EXPRESS schema may be made visible within the scope of another schema view via an interface specification as described in clause 13.

Syntax:

```
189  schema_view_decl = SCHEMA_VIEW schema_view_id ';' { reference_clause }
     [ constant_decl ] schema_view_body_element_list END_SCHEMA_VIEW ';' .
167  reference_clause = REFERENCE FROM schema_ref_or_rename [ '('
     resource_or_rename { ',' resource_or_rename } ')' ] [ AS ( SOURCE |
     TARGET ) ] ';' .
188  schema_view_body_element_list = schema_view_body_element {
     schema_view_body_element } .
187  schema_view_body_element = function_decl | procedure_decl | view_decl
     | rule_decl .
```

Rules and restrictions:

a) The syntax `AS ( SOURCE | TARGET )` shall not be used in view schema.

EXAMPLE **ap203_arm** names a schema_view that may contain declarations defining a view over the schema **config_control_design**.

```
SCHEMA_VIEW ap203_arm;
REFERENCE FROM config_control_design;
VIEW part_version ...
(* other declarations as appropriate *)
END_SCHEMA_VIEW;
```

9.6 Schema map declaration

A schema map declaration defines a common scope for a collection of related declarations. A schema map shall identify one or more schemas as source and one or more schemas as target (see 13.2). A schema map is not limited to declarations in language subsets (see Table 1).

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

Declarations in one schema map may be made visible within the scope of another schema map via an interface specification as described in clause 13.

Syntax:

```
184  schema_map_decl = SCHEMA_MAP schema_map_id ';' reference_clause {
     reference_clause } [ constant_decl ] schema_map_body_element_list
     END_SCHEMA_MAP ';' .
182  schema_map_body_element = function_decl | procedure_decl | view_decl |
     map_decl | dependent_map_decl | rule_decl .
```

Rules and restrictions:

a) The schema map shall include explicitly, or by reference using the `REFERENCE` language element (see 13.2), at least one map declaration.

EXAMPLE 1 **iges2step** names a schema_map that may contain declarations for translating geometry defined using an EXPRESS data set base upon IGES into a data set based on ISO 10303-203.

```
SCHEMA_MAP iges2step;
REFERENCE FROM step_schema AS TARGET;
REFERENCE FROM iges_express_schema AS SOURCE;
MAP iges_structure ...
(* other declarations as appropriate *)
END_SCHEMA_MAP;
```

A schema map may reference EXPRESS schema, other schema map schema and schema view schema through use of the `reference_clause` language syntax (see 13.2).

Syntax:

```
184  schema_map_decl = SCHEMA_MAP schema_map_id ';' reference_clause {
     reference_clause } [ constant_decl ] schema_map_body_element_list
     END_SCHEMA_MAP ';' .
167  reference_clause = REFERENCE FROM schema_ref_or_rename [ '('
     resource_or_rename { ',' resource_or_rename } ')' ] [ AS ( SOURCE |
     TARGET ) ] ';' .
186  schema_ref_or_rename = [ general_schema_alias_id ':' ]
     general_schema_ref .
```

Rules and restrictions:

a) A schema map shall reference at least one EXPRESS schema designated as a mapping source using the `AS SOURCE` syntax.

b) A schema map shall reference at least one EXPRESS schema designated as a mapping target using the `AS TARGET` syntax.

EXAMPLE 2 This example illustrates the designation of source and target EXPRESS schemas. EXPRESS schema **schema_target_one** is referenced as the target of mapping. EXPRESS schema **schema_source_one** is referenced as the source of mapping; it may be referred to as **s1** within the scope of this schema_map.


```
SCHEMA_MAP map_name;
  REFERENCE FROM schema_target_one AS TARGET;
  REFERENCE FROM s1 : schema_source_one AS SOURCE;
  MAP my_map AS ...
END_SCHEMA_MAP;
```

9.7 Local declaration

A map partition may declare local variables. Local variables are declared with the `LOCAL` language element. A local variable is only visible within the scope of the map partition in which it is declared, including its `IDENTIFIED_BY`, `WHERE` and `SELECT` language elements. When a map partition is evaluated, all local variable initially have an indeterminate value unless an initializer is explicitly given. Local variables may be assigned values in the body of the map partition and may be referenced in expressions.

Syntax:

```
 47  binding_header = [ PARTITION partition_id ';' ] [ from_clause ] [
     local_decl ] [ where_clause ] [ identified_by_clause ] [
     ordered_by_clause ] .
129  local_decl = LOCAL local_variable { local_variable } END_LOCAL ';' .
130  local_variable = variable_id { ',' variable_id } ':' parameter_type [
     ':=' expression ] ';' .
```

Rules and restrictions:

a) `local_decl` shall not be used in a view declaration.

EXAMPLE In this example a local variable is assigned a target entity data type instance value. An attribute of that entity is updated in the body of the map.

```
SCHEMA_MAP arm2aim;

REFERENCE FROM arm AS SOURCE;
REFERENCE FROM aim AS TARGET;

MAP product_map AS p : product;
  p : product;
 FROM
   ap : arm_product;
 LOCAL
   asc : applied_security_classification;
 END_LOCAL;
 SELECT
   asc := asc@security_classification_map(ap.security);
   asc.items := asc.items + p;
END_MAP;

MAP security_classification_map AS
  asc : applied_security_classification;
  scl : security_classification;
 FROM
   arm_asc : arm_security_classification;
 SELECT
   asc.items := [];
   asc.classification := scl;
END_MAP;

END_SCHEMA_MAP;
```


9.8 Constant declaration

Constants may be defined for use within the WHERE language element of a view or map declaration, or within the body of a map declaration or algorithm. Entity data type valued constants are made present in a target population if they are referenced by entity data type instances of that target population.

Constant declarations are as defined in ISO 10303-11, 9.4.

9.9 Function declaration

Functions may be defined for use within the WHERE language element of a view or map declaration, or within the body of a map declaration. An entity instance created and returned by the evaluation of a function is made present in a target population if it is referenced by an attribute of a map declaration target instance.

Function declarations are as defined in ISO 10303-11, 9.5.1.

9.10 Procedure declaration

Procedures may be defined for use within the bodies of function declarations. Procedures shall not be used directly within the body of a map or view declaration.

Procedure declarations are as defined in ISO 10303-11, 9.5.2.

9.11 Rule declaration

Rules may be defined for use within a schema view.

Rule declarations are as defined in ISO 10303-11, 9.6.

Inclusion of rule declarations has no effect on the execution model nor on the contents of source nor view extents.

10 Expressions

10.1 Overview

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

The built-in functions defined in Clause 15, and the operators defined in clause 12 of ISO 10303-11; 1994 apply to this part of ISO 10303. Arguments of view data types shall be treated as arguments of entity data types. The relationship between view definitions and entity definitions is defined in annex C.

Syntax:

```
 81  expression = simple_expression [ rel_op_extended simple_expression ] .
169  rel_op_extended = rel_op | IN | LIKE .
168  rel_op = '<' | '>' | '<=' | '>=' | '<>' | '=' | ':<>:' | ':=:' .
193  simple_expression = term { add_like_op term } .
216  term = factor { multiplication_like_op factor } .
 84  factor = simple_factor [ '**' simple_factor ] .
194  simple_factor = aggregate_initializer | entity_constructor |
     enumeration_reference | interval | query_expression | ( [ unary_op ] (
     '(' expression ')' | primary ) ) | case_expr | for_expr | if_expr .
158  primary = literal | ( qualifiable_factor { qualifier } ) .
163  qualifiable_factor = attribute_ref | constant_factor | function_call |
     general_ref | map_call | population | target_parameter_ref |
     view_attribute_ref | view_call .
```

Entity constructors create instances that are local only to the function or procedure in which they are used. Instances produced by entity constructors shall not create target nor source populations.

Evaluation of an expression is governed by the precedence of the operators which form part of the expression. Expressions enclosed by parentheses are evaluated before being treated as a single operand. Evaluation proceeds from left to right, with the highest precedence being evaluated first. Table 3 specifies the precedence rules for all of the operators of EXPRESS-X. Operators in the same row have the same precedence, and the rows are ordered by decreasing precedence. An operand between two operators of different precedence is bound to the operator with the higher precedence. An operand between two operators of the same precedence is bound to the one on the left.

Table 3: Operator precedence

Precedence	Description	Operators
1	Component Reference	[] . \ :: <- {}
2	Unary Operators	+ - NOT
3	Exponentiation	**
4	Multiplication/Division	/ * DIV MOD AND ||
5	Addition/Subtraction	+ - OR XOR
6	Relational	= <> <= >= < > :=: :<>: IN LIKE

10.2 View call

A view call is an expression that evaluates to a view data type instance or aggregate of view data type instances. The view call provides a means to access a view data type instance (or instances) via the view name and actual parameters corresponding to its binding instance (when no IDENTIFIED_BY is defined by the view) or IDENTIFIED_BY language element expression values (when IDENTIFIED_BY is defined by the view). When no IDENTIFIED_BY language element is present in the partition, the number, type, and order of the actual parameters shall agree with those of the source parameters of the FROM language element in the view. When an IDENTIFIED_BY language element is present, the number, type and order of the actual parameters shall agree with those of the expressions of the IDENTIFIED_BY language element.

If no binding instance corresponds to the actual parameters of the view call, the call evaluates to an indeterminate value.

A view call identifies a single partition of a view. If the view contains more than one partition, a partition_qualification may be specified to explicitly select the partition for which a value shall be returned. Alternatively, when no partition_qualification is specified, the partition for which a value is returned shall be the first partition to appear lexically in the declaration for which the number and type of arguments agree with the parameters and the WHERE language element returns true. For type agreement, the type compatibility rules of ISO 10303-11 clause 12.11 shall apply.

A view call referencing a constant partition shall have an empty argument list.

Syntax:

```
227  view_call = view_reference [ partition_qualification ] '(' [
     expression_or_wild { ',' expression_or_wild } ] ')' .
153  partition_qualification = '\' partition_ref .
 82  expression_or_wild = expression | '_' .
```


EXAMPLE This example illustrates the use of a view call to define a relationship between two view data types. The IDENTIFIED_BY language element in the **person_part** partition specifies one expression, a.creator. View calls to **approver\person_part** will therefore be supplied with one argument, a STRING which is also the creator attribute of an approval entity instance.

```
SCHEMA_VIEW example;
REFERENCE FROM src_schema;
VIEW approver;
 PARTITION person_part;
  FROM a : approval; p : person;
  WHERE a.creator = p.name;
  IDENTIFIED_BY a.creator;
  SELECT
    approver_id : INTEGER := p.id;
 PARTITION org_part;
  FROM a : approval; o : organization;
  WHERE a.creator = o.name;
  IDENTIFIED_BY a.creator;
  SELECT
    approver_id : INTEGER := o.id;
END_VIEW;

VIEW design_order;
  FROM a : approval;
  SELECT
    id : STRING := a.id;
    approved_by : approver :=
                  approver\person_part(a.creator);
END_VIEW;
END_SCHEMA_VIEW;

SCHEMA src_schema;

ENTITY approval;
  id : STRING;
  creator : STRING;
END_ENTITY;

ENTITY person;
  name : STRING;
  id : INTEGER;
END_ENTITY;
ENTITY organization;
 id : INTEGER;
 name : STRING;
END_ENTITY;

END_SCHEMA;

(* Source data set in ISO 10303-21 form – see [2] *)
#1=APPROVAL('a_1','Jones');
#2=APPROVAL('a_2','Smith');
#3=APPROVAL('a_3','Jones');
#4=PERSON('Jones',123);
#5=PERSON('Smith',234);
(* Resulting view instances in ISO 10303-21 form *)
#101=APPROVER(123);
#102=APPROVER(234);
#103=DESIGN_ORDER('a_1',#101);
#104=DESIGN_ORDER('a_2',#102);
#105=DESIGN_ORDER('a_3',#101);
```


10.3 Map call

A map call is an expression that evaluates to target entity instances or an aggregate of entity instances. The map call provides a means to access the values produced by evaluation of a map (including dependent map) via the map name and actual parameters corresponding to its binding instance (when no IDENTIFIED_BY is defined by the map) or IDENTIFIED_BY language element expression values (when IDENTIFIED_BY is defined by the map). When no IDENTIFIED_BY language element is present in the map, then the number, type, and order of the actual parameters shall agree with those of the source parameters of the FROM language element in the partition. When an IDENTIFIED_BY language element is present, the number, type and order of the actual parameters shall agree with those of the expressions of the IDENTIFIED_BY language element.

If no binding instance corresponds the actual parameters of the map call, the call evaluates to an indeterminate value.

A map call identifies a single partition of a map. If the map contains more than one partition a partition_qualification may be provided to explicitly select the partition for which a value shall be returned. Alternatively, when no partition_qualification is specified, the partition for which a value is returned shall be the first partition that appears lexically in the map declaration for which the number and type of arguments agree with the parameters and the WHERE language element returns true. For type agreement, the type compatibility rules of ISO 10303-11 clause 12.11 shall apply.

Syntax:

```
135  map_call = [ target_parameter_ref '@' ] map_reference [
     partition_qualification ] '(' expression_or_wild { ','
     expression_or_wild } ')' .
153  partition_qualification = '\' partition_ref .
```

Rules and restrictions:

a) target_parameter_ref shall refer to a parameter reference declared in the map declaration referenced as map_reference.

b) If the map declaration referenced by the map call declares more than one target parameter, the target_parameter_ref @ syntax shall be used to identify the target to be returned by the map call.

EXAMPLE 1 This example illustrates the use of a map call to define a relationship between entities in the target schema.

```
(* source schema *)
SCHEMA src;
ENTITY approval;
  id : STRING;
  creator : STRING;
END_ENTITY;
END_SCHEMA;

(* target schema *)
```



```
SCHEMA tar;
ENTITY person;
  name : STRING;
END_ENTITY;
ENTITY design_order;
  id : STRING;
  approved_by : person;
END_ENTITY;
END_SCHEMA;

SCHEMA_MAP example;
REFERENCE FROM src AS SOURCE;
REFERENCE FROM tar AS TARGET;

MAP person_map AS p : tar.person;
FROM a : approval;
IDENTIFIED_BY a.creator;
SELECT
  p.id := a.creator;
END_MAP;

MAP design_order_map AS d : tar.design_order;
FROM a : approval;
SELECT
  d.id := a.id;
  d.approved_by := p@person_map(a.creator); -- map call
END_MAP;
END_SCHEMA_MAP;

(* source instance set written as ISO 10303-21 instances – see [2] *)
#1 = APPROVAL('a_1','Miller');
#2 = APPROVAL('a_2','Jones');
#3 = APPROVAL('a_3','Miller');

(* Resulting target instances in ISO 10303-21 form (see [2]). *)
#101=PERSON('Miller');
#102=PERSON('Jones');
#103=DESIGN_ORDER('a_1',#101);
#104=DESIGN_ORDER('a_2',#102);
#105=DESIGN_ORDER('a_3',#101);
```

EXAMPLE 2 This example illustrates the use of map calls where partitions are not explicitly identified.

```
SCHEMA source_schema;
TYPE person_select = SELECT (male, female, child);
END_TYPE;

ENTITY male;
 name: STRING;
END_ENTITY;

ENTITY female;
 name : STRING;
END_ENTITY;

ENTITY child;
 name : STRING;
 parents : SET of person_select;
END_ENTITY;
END_SCHEMA;

SCHEMA target_schema;
ENTITY person;
 name : STRING;
END_ENTITY;
```



```
ENTITY person_with_parents
 SUBTYPE OF (person);
 parents : SET [1:2] of person;
END_ENTITY;
END_SCHEMA;

SCHEMA_MAP source_to_target;
REFERENCE FROM source_schema AS SOURCE;
REFERENCE FROM target_schema AS TARGET;

MAP person_map AS
 p : person;
PARTITION p_male;
 FROM m : male;
 SELECT
  p.name := m.name;
PARTITION p_female;
 FROM f : female;
 SELECT
  p.name := f.name;
PARTITION p_child;
 FROM c : child;
 RETURN person_with_parents_map(c);
END_MAP;

MAP person_with_parents_map AS
 p : person_with_parents;
 FROM c: child;
 WHERE SIZEOF (c.parents) > 0; -- only create person_with_parents if there is
                               -- at least one parent.
 SELECT
  p.name := c.name;
  p.parents := FOR EACH par IN c.parents;
                  RETURN person_map(par); -- see 10.5.
END_MAP;
END_SCHEMA_MAP;

(* example population:
SOURCE:

#1=FEMALE('Julia');
#3=MALE('Richard');
#4=CHILD('Mary', (#1));
#5=CHILD('Paul', (#1,#3));

TARGET:
#6=PERSON('Julia');
#7=PERSON_WITH_PARENTS('Mary', (#6));
#8=PERSON('Richard');
#9=PERSON_WITH_PARENTS('Paul', (#8,#6));

*)
```


10.4 Partial binding calls

A partial binding call is a view or map call in which one or more arguments is the '_' language element. The partial binding call evaluates to a subset of the view extent or target population of the view or map declaration respectively.

The FROM language element defines the elements of binding instances in the binding extent of the view or map call (see 9.2.1). In a view or map declaration not specifying an IDENTIFIED_BY language element, the view or map declaration defines a functional correspondence from binding instances to instances in the view extent or target population: there is a unique instance in the view extent or target population for each binding instance. In view or map declarations specifying an IDENTIFIED_BY

language element this correspondence is not functional, there is not a unique instance in the view extent or target population for each binding instance, rather there is a unique instance in the view extent or target population for each equivalence class of binding instances defined by the IDENTIFIED_BY language element (see 9.2.2).

When IDENTIFIED_BY is not specified, each qualified binding instance serves as a key to identify an instance of the correspondence. When IDENTIFIED_BY is specified, any binding instance in the equivalence class serves as a key to identify an instance of the correspondence. The elements (components of the keys) of these two kinds of keys are defined by the FROM language element and IDENTIFIED_BY expressions respectively.

A partial binding call is a view or map call in which a value for one or more of the elements of these keys is not specified, (designated by '_' language element in the syntax).

When IDENTIFIED_BY is not specified, this partially specified key corresponds to the set of qualified binding instances that match on the specified elements of the key. The components of the key that are not specified (designated by '_') are ignored in the matching process.

When IDENTFIED_BY is specified, this partially specified key is matched against values from the evaluation of the corresponding expression in the IDENTIFIED_BY language element. The components of the key that are not specified (designated by '_') are ignored in the matching process.

The value of the partial binding call is the set of view instances (view call) or entity instances (map call) corresponding to the binding instances collected by these means.

EXAMPLE In the following, the various versions associated with a part are collected by using a partial binding call. Returned by the partial binding call **version_and_its_product** is the subset of the extent for which the second element of the binding instance is instance-equal to the specified **product** instance.

```
VIEW part;
FROM p : product;
SELECT
  versions : SET OF version_and_its_product
     := version_and_its_product(_, p);
END_VIEW;

VIEW version_and_its_product;
FROM pdf : product_definition_formation; p : product;
WHERE p :=: pdf.of_product;
SELECT
  the_version : product_definition_formation := pdf;
END_VIEW;
```

10.5 FOR expression

A FOR expression computes an aggregate value by iterating over an index value or aggregate. This specification defines two forms of iteration:

– the foreach_expr expression iterates over an EXPRESS aggregate and evaluates the expression for each element of that aggregate. The result of these evaluations are collected in an aggregate which is the value of the FOR expression;

– the forloop_expr expression is controlled by the repeat_control (ISO 10303-11, 13.9) and evaluates the expression for each iteration.

If no iteration occurs, the FOR expression evaluates to an empty aggregate.

Syntax:

```
 89  for_expr = FOR ( foreach_expr | forloop_expr ) .
 85  foreach_expr = EACH variable_id IN expression [ where_clause ] RETURN
     expression .
 86  forloop_expr = repeat_control RETURN expression .
171  repeat_control = [ increment_control ] [ while_control ] [
     until_control ] .
113  increment_control = variable_id ':=' bound_1 TO bound_2 [ BY increment
     ] .
232  while_control = WHILE logical_expression .
222  until_control = UNTIL logical_expression .
```

Rules and restrictions:

a) Each `expression` of the `foreach_expr` shall evaluate to an EXPRESS aggregate, extent, or view extent.

b) A view declaration shall not contain the `FOR` expression.

The `foreach_expr` language element implicitly declares an iterator variable `variable_id` (see syntax rule 85) of type `GENERIC` visible within the scope of the `where_clause` and `expression`. The expression after the `IN` language element shall be the EXPRESS aggregate over which the iteration is performed. In each iteration of the loop, an element of the aggregate is bound to this iterator variable. The elements are bound in order proceeding from `LOINDEX` to `HIINDEX` (ISO 10303-11, 15.17 and 15.11).

The `RETURN` language element of syntax rule 85 specifies an expression evaluated for each element of the iteration. The expression is evaluated in an environment binding the iterator variable to each value of the source aggregate in turn. The result of each evaluation is added to the target attribute aggregate as though the union operator (ISO 10303-11, 12.6.3) were applied with the target attribute aggregate as its left operand and the result of the expression evaluation as its right operand.

The optional `where_clause` of syntax rule 85 specifies an expression that shall return a `LOGICAL` or indeterminate value. The expression following the `RETURN` language element is only evaluated and the value included in the result aggregate if the `where_clause` evaluates to `TRUE`.

EXAMPLE 1 In this example, the target entity **component** maps to the source entity **product_definition** and all instances of **product_definition_name** which reference one instance of **product_definition** are grouped into the target attribute **component.names**. This is specified as follows.

```
(* Source schema *)
SCHEMA source_schema;
ENTITY product_definition;
  product_name : STRING;
  description : STRING;
END_ENTITY;
ENTITY product_definition_name;
  name    : STRING;
  of_product_definition : product_definition;
END_ENTITY;
END_SCHEMA;
(* Target schema *)
SCHEMA target_schema;
ENTITY component;
  names : SET [0:?] OF STRING;
  product_name  : STRING;
  description  : STRING;
END_ENTITY;
END_SCHEMA;

(* Mapping definition *)
SCHEMA_MAP example;
REFERENCE FROM source_schema AS SOURCE;
REFERENCE FROM target_schema AS TARGET;
MAP component_map AS c : component;
FROM pd : product_definition;
SELECT
  c.description := pd.description;
  c.product_name := pd.product_name;
  c.names := FOR EACH pdn_instance
               IN  EXTENT('SOURCE_SCHEMA.PRODUCT_DEFINITION_NAME');
               WHERE pdn_instance.of_product_definition :=: pd;
           RETURN pdn_instance.name;
END_MAP;
END_SCHEMA_MAP;
```

EXAMPLE 2 This example illustrates the use of nested FOR expressions. Example 1 is extended as follows.

```
(* Source schema *)
SCHEMA source_schema;

ENTITY product_definition;
  product_name : STRING;
  description : STRING;
END_ENTITY;
ENTITY product_definition_name;
  name : STRING;
  of_product_definition : product_definition;
END_ENTITY;
ENTITY product_definition_value;
  of_pdn : product_definition_name;
  val : STRING;
END_ENTITY;
END_SCHEMA;
```



```
(* Target schema *)
SCHEMA target_schema;
ENTITY component;
  values : SET [0:?] OF SET [0:?] OF STRING;
  product_name  : STRING;
  description  : STRING;
END_ENTITY;
END_SCHEMA;
```

All instances of **product_definition_value** which reference one instance of **product_definition_name** are grouped together and are assigned to the inner aggregate of component.values. This is specified as follows.

```
(* Mapping definition *)
SCHEMA_MAP example;

REFERENCE FROM source_schema AS SOURCE;
REFERENCE FROM target_schema AS TARGET;

MAP component_map AS c : component;
FROM pd : product_definition;
SELECT
  c.description := pd.description;
  c.product_name := pd.product_name;
  c.values := FOR EACH pdn_instance
                IN EXTENT('SOURCE_SCHEMA.PRODUCT_DEFINITION_NAME');
                WHERE pdn_instance.of_product_definition :=: pd;
            RETURN FOR EACH pdv_instance
                        IN EXTENT('SOURCE_SCHEMA.PRODUCT_DEFINITION_VALUE');
                     WHERE pdv_instance.of_pdn :=: pdn_instance;
                    RETURN pdv_instance.val;
END_MAP;
END_SCHEMA_MAP;
```

10.6 IF expression

The IF language element provides for the conditional evaluation of expression language elements based on the evaluation of logical_expression of syntax rule 110. If no logical_expression is satisfied and an ELSE language element is not specified, the map or view attribute is assigned an indeterminate value.

Syntax:

```
110  if_expr = IF logical_expression THEN expression { ELSIF
     logical_expression expression } [ ELSE expression ] END_IF .
```

10.7 CASE expression

The CASE language element provides for the conditional evaluation of expression language elements following the pattern of the EXPRESS CASE statement (ISO 10303-11, 13.4). If no OTHERWISE language element is specified and no case_expr_action is satisfied the map or view attribute is assigned an indeterminate value.

Syntax:

```
 56  case_expr = CASE selector OF { case_expr_action } [ OTHERWISE ':'
     expression ] END_CASE .
 57  case_expr_action = case_label { ',' case_label } ':' expression ';' .
```


Rules and restrictions:

a) The CASE expression shall not be used in conformance class 1 conforming schema views.

EXAMPLE This example illustrates use of the CASE expression.

```
SCHEMA source_schema;
ENTITY approval;
 status : STRING;
END_ENTITY;
END_SCHEMA;

SCHEMA target_schema;
ENTITY my_approval;
 status : INTEGER;
END_ENTITY;
END_SCHEMA;

SCHEMA_MAP mapping_example;

REFERENCE FROM source_schema AS SOURCE;
REFERENCE FROM target_schema AS TARGET;


MAP approval_map AS ma : my_approval;
FROM a : approval;
SELECT
  ma.status := CASE a.status OF
                 'approved'     : 1;
                 'not_yet_approved' : -1;
                 'disapproved'  : 0;
                 OTHERWISE      : 2;
               END_CASE;
END_MAP;
END_SCHEMA_MAP;
```

10.8 Forward path operator

The forward path operator (::) provides an aggregate of entity instances referenced by the value of attribute_ref in syntax rule 88. If the optional extent_reference of syntax rule 154 is specified, the result aggregate shall contain only entity instances of the type corresponding to extent_reference or of one of its subtypes.

Syntax:

```
 88  forward_path_qualifier = '::' attribute_ref [ path_condition ] .
154  path_condition = '{' extent_reference [ '|' logical_expression ] '}' .
```

Rules and restrictions:

a) The forward path operator shall not be used in conformance class 1 conforming schema views;

b) A variable having the same name as extent_reference is implicitly declared within the scope of the forward path expression.

NOTE The variable does not have to be declared elsewhere, and it does not persist outside the expression.

NOTE The example of clause 10.9 illustrates the use of the forward path operator.

When logical_expression of syntax rule 154 is specified, elements are taken in turn from the referenced extent and are bound to the implicitly declared variable. The logical_expression is then

evaluated in the environment of this binding. For each such binding of the variable, if `logical_expression` evaluates to TRUE the element is added to the result; otherwise, it is not.

NOTE The unnest function referenced below accepts one argument of arbitrary type (including a nested aggregate) and returns an aggregate whose elements are non-aggregate types. For example, unnest([[a],[b,c],[[d]]]) returns [a,b,c,d]. See annex E for a definition of the unnest function.

EXAMPLE For some population **a**, an entity reference **product** and an attribute of instances in the extent **a**, **of_product**, the expression **result := a::of_product{product}** is equivalent to the following EXPRESS specification:

```
LOCAL
  result : AGGREGATE OF GENERIC := [];
  tmp : AGGREGATE OF GENERIC := [];
END_LOCAL;
tmp := unnest(a);
REPEAT i := 1 TO HIINDEX(tmp);
  result := result + QUERY(e <* unnest(tmp[i].of_product)|
                    'SCHEMA_NAME.PRODUCT' IN TYPEOF(e));
END_REPEAT;
result := unnest(result);
```

The expression `result := a::x` is equivalent to the EXPRESS specification:

```
result := [];
tmp := unnest(a);
REPEAT i := 1 TO HIINDEX(tmp);
  result := result + unnest(tmp[i].x);
END_REPEAT;
result := unnest(result)
```

10.9 Backward path operator

The backward_path_operator (<-) provides an aggregate of entity instances using the expression on the right side of the operator. The expression `a <-x { b | c }` evaluates to an aggregate of entity instances such that for each element **e** of the aggregate:

– the attribute **x** of **e** references an element of the unnested **a**;

– **e** is of type **b**;

– and, the logical expression **c** evaluate to `TRUE` in an environment where the implicit variable **b** is bound to **e**.

Syntax:

```
 43 backward_path_qualifier = '<-' [ attribute_ref ] path_condition .
154 path_condition = '{' extent_reference [ '|' logical_expression ] '}' .
```

Rules and restrictions:

a) The backward path opeartor shall not be used in conformance class 1 conforming schema views;

b) `attribute_ref` shall be defined in some partial entity data type of each instance of the argument extent.

c) A variable having the same name as `extent_reference` is implicitly declared within the scope of the backward path expression.

When identifier `a` represents an population, the expression `result := a<-x{b}` is equivalent to the EXPRESS specification:

```
LOCAL
  result : AGGREGATE OF GENERIC := [];
  tmp : AGGREGATE OF GENERIC := [];
END_LOCAL;
result := [];
tmp := unnest(a);
REPEAT i := 1 TO HIINDEX(tmp);
  result := result +  QUERY(e <* USEDIN(tmp[i], '')  |
            ('SCHEMA_NAME.B' IN TYPEOF(e))
            AND (tmp[i] IN e.x));
END_REPEAT;
```

The expression `a<-{b}` is equivalent to the EXPRESS specification:

```
result := [];
tmp := unnest(a);
REPEAT i := 1 TO HIINDEX(tmp);
   result := result + QUERY(e <* USEDIN(tmp[i],'') |
                        ('SCHEMA_NAME.B' IN TYPEOF(e)));
END_REPEAT;
```

NOTE The unnest function referenced above accepts one argument of arbitrary type (including a nested aggregate) and returns an aggregate whose elements are non-aggregate types. for example, unnest([[a],[b,c],[[d]]]) returns [a,b,c,d]. See annex E for a definition of the unnest function.

EXAMPLE In this example path operators are used to compute the source aggregate of an instantiation loop. The source aggregate contains all instances of type **document_file** referring to a **representation_type** instance with **name** of 'digital' and are referenced as **documentation_ids** of a **product_definition_with_associated_documents** instance which refers to the source **product_definition_formation** instance.

```
SCHEMA document_schema;

ENTITY folder;
   name : STRING;
END_ENTITY;

ENTITY file;
    name     : STRING;
    location : folder;
END_ENTITY;
END_SCHEMA;

SCHEMA source_schema;

ENTITY product_definition_formation;
 id : STRING;
 name : STRING;
END_ENTITY;

ENTITY product_definition_with_associated_documents;
 documentation_ids : SET OF document_file;
 formation : product_definition_formation;
END_ENTITY;

ENTITY document_file;
 name : STRING;
 representation_type : document_representation;
END_ENTITY;

ENTITY document_representation;
 name : STRING;
END_ENTITY;

END_SCHEMA;
```



```
SCHEMA_MAP example2;
REFERENCE FROM document_schema AS TARGET;
REFERENCE FROM source_schema AS SOURCE;
MAP document_map AS
  folder : folder;
  files:  AGGREGATE OF file;
  FROM pdf : product_definition_formation;
    FOR EACH f IN
      pdf<-formation{product_definition_with_associated_documents}
            ::documentation_ids{document_file
            | document_file.representation_type.name = 'digital'}
     INDEXING i;
     SELECT
       files[i].name := f.name;
       files[i].location := folder;
       folder.name := pdf.name;
END_MAP;
END_SCHEMA_MAP;
```

11 Built-in functions

11.1 Extent - general function

```
FUNCTION EXTENT ( R : STRING ) : SET OF GENERIC;
```

The EXTENT function evaluates to a set of all instances in the population that are of the type specified by the parameter.

Parameters:

a) R is a string that contains the name of a entity data type or view data type. Such names are qualified by the name of the schema which contains the definition of the type.

Result: A set containing all instances of the entity data type or view data type specified by the parameter. It is an error to specify as the parameter a type which is neither a view data type nor an entity data type defined in a source schema.

EXAMPLE The extent of the **action** entity data type in the **automotive_design** schema is:

```
EXTENT('AUTOMOTIVE_DESIGN.ACTION');
```


12 Scope and visibility

12.1 Overview

An EXPRESS-X declaration creates an identifier that can be used to reference the declared item in other parts of the schema view / schema map (or in other schema views / schema maps). Some EXPRESS-X constructs implicitly declare items, attaching identifiers to them. An item is said to be vis-

ible in those areas where an identifier for a declared item may be referenced. An item shall only be referenced where its identifier is visible.

Certain EXPRESS-X items define a region (block of text) called the scope of the item. This scope limits the visibility of identifiers declared within it. Scope can be nested; that is, an EXPRESS-X item which establishes a scope may be included within the scope of another item. There are constraints on which items may appear within the scope of a particular EXPRESS-X item. These constraints are specified in terms of language element syntax.

For each of the items specified in Table 4 below, 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. For language elements defined in EXPRESS (ISO 10303-11) the limits of the scope defined and the visibility of the declared identifier is as specified in EXPRESS (see ISO 10303-11, 10.3).

The general rules of scope and visibility of EXPRESS apply (see ISO 10303-11, 10.1, 10.2, 10.2.1).

Table 4: Scope and identifier defining items

Item	Scope	Identifier
Instantiation Loop	•	•[1]
For Expression	•	•[1]
Dependent map, Map	•	•
Dependent view, View	•	•
Source Parameter		•
Target Parameter		•
Path Expression	•	•
NOTE 1 - the identifier is an implicitly declared variable within the scope of the declaration.		

12.2 Schema view

Visibility: A schema view declaration is visible to all other schema views.

Scope: A schema view declaration defines a new scope. This scope extends from the keyword `SCHEMA_VIEW` to the keyword `END_SCHEMA_VIEW` that terminates that schema view declaration.

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

– constant;

– function;

– procedure;

– rule;

– view.

12.3 Schema map

Visibility: A schema map identifier is visible to all other schema views and schema maps.

NOTE The map declarations of a schema map shall not be referenced by a schema view.

Scope: A schema map declaration defines a new scope. The scope extends from the SCHEMA_MAP keyword to the keyword END_SCHEMA_MAP which terminates the schema map declaration.

Declarations: The following items may declare identifiers within the scope of a schema map:

– constant;

– function;

– procedure;

– view;

– dependent view;

– map;

– dependent map;

– rule.

12.4 View and dependent view

Visibility: A view or dependent view identifier is visible in the scope of the schema view, or schema map in which it is declared. A view or dependent view identifier remains visible within inner scopes that do not redeclare that identifier.

Scope: A view or dependent view declaration defines a new scope. This scope extends from the keyword VIEW (DEPENDENT_VIEW) to the keyword END_VIEW (END_DEPENDENT_VIEW) which terminates that view declaration.

Declarations: The following items may declare identifiers within the scope of a view declaration:

– partition label;

– FROM language element;

– IDENTIFIED_BY language element;

– view attribute.

12.5 View partition label

Visibility: The visibility rules of a view partition label are identical with the visibility rules of the view (see 12.4).

12.6 View attribute identifier

Visibility: A view attribute identifier is visible in the scope of the view in which it is declared.

12.7 FOR expression

Visibility: The implicitly declared iterator variable is visible in the expression following the RETURN keyword of the FOR expression and in the WHERE language element of syntax rule 85.

Scope: A FOR expression defines a new scope. This scope is visible for the expression followed by the RETURN keyword of the FOR expression.

Declarations: The following items may declare identifiers within the scope of a FOR expression:

– FOR expression;

– QUERY expression.

12.8 Map and dependent map

Visibility: A map or dependent map identifier is visible in the scope of the schema view, or schema map in which it is declared. A map or dependent map identifier remains visible within inner scopes that do not redeclare that identifier.

Scope: A map or dependent map declaration defines a new scope. This scope extends from the keyword MAP (DEPENDENT_MAP) to the keyword END_MAP (END_DEPENDENT_MAP) which terminates the map declaration.

Declarations: The following items may declare identifiers within the scope of a map or dependent map declaration:

– FOR expression;

– QUERY expression;

– instantiation loop;

– FROM language element;

– IDENTIFIED_BY language element;

– attribute.

12.9 FROM Language Element

Visibility: A source parameter identifier is visible in the partition in which it is declared and in subtype views or maps of that view or map.

12.10 Instantiation Loop

Visibility: The implicitly declared identifiers of an instantiation loops are visible within the items of the SELECT language element followed by the instantiation loop. The scope is terminated either by the END_MAP keyword or by the preceding instantiation loop.

Declarations: The QUERY expression may declare identifiers within the scope of an instantiation loop.

12.11 Path expression

Visibility: Within a path condition, the attributes of the type qualifier and its supertypes are visible as well as the SELF keyword. In addition all identifiers declared outside the expression are visible.

Scope: The path expression defines a new nesting scope within the path_condition language element, which is the scope of the entity data type of the type qualifier (see ISO-10303-11, 10.3.5). The new scope starts with the '|' syntax element and ends with the terminating closing brace '}'.

Declarations: The QUERY expression may declare identifiers within the scope of a path condition.

13 Interface specification

13.1 Overview

Interface specifications enable items declared in one foreign schema, view schema and map schema to be visible in another view schema or map schema.

13.2 The REFERENCE language element

The REFERENCE language element enable items declared in one schema, schema view or schema map to be visible in the current schema view or schema map.

A REFERENCE specification makes the following items, declared in a foreign schema, schema view, or schema map, to be visible in the current schema view:

– view

– dependent view;

– map

– dependent map;

– constant;

– entity;

– type;

– function;

– procedure;

– rule.

A REFERENCE specification identifies the name of the foreign schema, schema view or schema map, and optionally the names of EXPRESS or EXPRESS-X items declared therein. If there are no names specified, all the items declared in the foreign schema, schema view, or schema map are visible within the current schema view or schema map.

Syntax:

```
167  reference_clause = REFERENCE FROM schema_ref_or_rename [ '('
     resource_or_rename { ',' resource_or_rename } ')' ] [ AS ( SOURCE |
     TARGET ) ] ';' .
186  schema_ref_or_rename = [ general_schema_alias_id ':' ]
     general_schema_ref .
103  general_schema_ref = schema_ref | schema_map_ref | schema_view_ref .
174  resource_or_rename = resource_ref [ AS rename_id ] .
```

Rules and restrictions:

a) The `rename_id` shall be unique within the scope of the referencing schema, including all other referenced identifiers. The referencing schema shall refer to the declaration using its `rename_id`.

b) The `general_schema_ref` shall be unique within the scope of the referencing schema.

c) A schema view shall not reference a map declaration.

EXAMPLE This example illustrates the designation of a source EXPRESS schema and the reference of resources from other schema. The resource **your_view_decl** is referenced from the schema **other_map_schema** and is renamed **my_view_decl** for use within this schema_view.

```
SCHEMA_VIEW my_view_schema;
   REFERENCE FROM automotive_design;
   REFERENCE FROM other_map_schema (your_view_decl AS my_view_decl);
END_SCHEMA_VIEW;
```


Annex A
(normative)

Information object registration

To provide for unambiguous identification of an information object in an open system, the object identifier

{ iso standard 10303 part(14) version(1) }

is assigned to this part of ISO 10303. The meaning of this value is defined is ISO/IEC 8824-1, and is described in ISO 10303-1.

Annex B
(normative)

EXPRESS-X language syntax

This annex defines the lexical elements of the language and the grammar rules that these elements shall obey.

NOTE This syntax definition will result in ambiguous parsers if used directly. It has been written so as to convey information regarding the use of identifiers. The interpreted identifiers define tokens that are references to declared identifiers, and therefore should not resolve to `simple_id`. This requires a parser developer to enable identifier reference resolution and return the required reference token to a grammar rule checker.

All of the grammar rules of EXPRESS specified in annex A of ISO 10303-11:1994 are also grammar rules of EXPRESS-X. In addition to the EXPRESS grammar rules, the grammar rules specified in the remainder of this annex are grammar rules of EXPRESS-X.

B.1 Tokens

The following rules specify the tokens used in EXPRESS-X. Except where explicitly stated in the syntax rules, no white space or remarks shall appear within the text matched by a single syntax rule in the following clauses.

B.1.1 Keywords

This subclause gives the rules used to represent the keywords of EXPRESS-X.

NOTE This subclause follows the typographical convention that each keyword is represented by a syntax rule whose left hand side is that keyword in uppercase.

The reserved words of EXPRESS-X are the reserved words of EXPRESS and the keywords and the names of built-in functions of EXPRESS-X. EXPRESS-X reserved words shall not be used as identifiers.

```
DEPENDENT_MAP = 'dependent_map' .
EACH = 'each' .
ELSIF = 'elsif' .
END_DEPENDENT_MAP = 'end_dependent_map' .
END_MAP = 'end_map' .
END_SCHEMA_MAP = 'end_schema_map' .
END_SCHEMA_VIEW = 'end_schema_view' .
END_VIEW = 'end_view' .
EXTENT = 'extent' .
IDENTIFIED_BY = 'identified_by' .
INDEXING = 'indexing' .
MAP = 'map' .
ORDERED_BY = 'ordered_by' .
PARTITION = 'partition' .
SCHEMA_MAP = 'schema_map' .
```



```
16  SCHEMA_VIEW = 'schema_view' .
17  SOURCE = 'source' .
18  TARGET = 'target' .
19  VIEW = 'view' .
```

B.1.2 Character classes

```
20  digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' .
21  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' .
22  simple_id = letter { letter  | digit  | '_' } .
```

B.1.3 Interpreted identifiers

NOTE All interpreted identifiers of EXPRESS are also interpreted identifiers of EXPRESS-X. The following are additional interpreted identifiers of EXPRESS-X.

```
23  partition_ref = partition_id .
24  map_ref =  map_id .
25  schema_map_ref = schema_map_id .
26  schema_view_ref = schema_view_id .
27  source_parameter_ref = source_parameter_id .
28  source_schema_ref = schema_ref .
29  target_parameter_ref =  target_parameter_id .
30  target_schema_ref = schema_ref .
31  view_attribute_ref = view_attribute_id .
32  view_ref = view_id .
```

B.2 Grammar rules

The following rules specify how the previous lexical elements and the lexical elements of EXPRESS may be combined into constructs of EXPRESS-X. White space and EXPRESS remarks may appear between any two tokens in these rules. The primary syntax rule for EXPRESS-X is `syntax_x`.

```
33  abstract_supertype_declaration = ABSTRACT SUPERTYPE [
    subtype_constraint ] .
34  actual_parameter_list = '(' parameter { ',' parameter } ')' .
35  add_like_op = '+' | '-' | OR | XOR .
36  aggregate_initializer = '[' [ element { ',' element } ] ']' .
37  aggregate_source =  simple_expression .
38  aggregate_type =  AGGREGATE [ ':' type_label ] OF parameter_type .
39  aggregation_types =  array_type | bag_type | list_type | set_type .
40  algorithm_head = { declaration } [ constant_decl ] [ local_decl ] .
41  array_type =  ARRAY bound_spec OF [ OPTIONAL ] [ UNIQUE ] base_type .
42  assignment_stmt =  general_ref { qualifier } ':=' expression ';' .
43  backward_path_qualifier = '<-' [ attribute_ref ] path_condition .
44  bag_type =  BAG [ bound_spec ] OF base_type .
```



```
45 base_type =  aggregation_types | simple_types | named_types .
46 binary_type =  BINARY [ width_spec ] .
47 binding_header = [ PARTITION partition_id ';' ] [ from_clause ]
   [ local_decl ] [ where_clause ] [ identified_by_clause ]
   [ ordered_by_clause ] .
48 boolean_type =  BOOLEAN .
49 bound_1 =  numeric_expression .
50 bound_2 =  numeric_expression .
51 bound_spec = '[' bound_1 ':' bound_2 ']' .
52 built_in_constant =  CONST_E | PI | SELF | '?' .
53 built_in_function =  ABS | ACOS | ASIN | ATAN | BLENGTH | COS | EXISTS
   | EXTENT | EXP | FORMAT | HIBOUND | HIINDEX | LENGTH | LOBOUND
   | LOINDEX | LOG | LOG2 | LOG10 | NVL | ODD | ROLESOF | SIN | SIZEOF
   | SQRT | TAN | TYPEOF | USEDIN | VALUE | VALUE_IN | VALUE_UNIQUE .
54 built_in_procedure =  INSERT | REMOVE .
55 case_action =  case_label { ',' case_label } ':' stmt .
56 case_expr =  CASE selector OF { case_expr_action } [ OTHERWISE ':'
   expression ] END_CASE .
57 case_expr_action =  case_label { ',' case_label } ':' expression ';' .
58 case_label =  expression .
59 case_stmt =  CASE selector OF { case_action } [ OTHERWISE ':' stmt ]
   END_CASE ';' .
60 compound_stmt =  BEGIN stmt { stmt } END ';' .
61 constant_body =  constant_id ':' base_type ':=' expression ';' .
62 constant_decl =  CONSTANT constant_body { constant_body } END_CONSTANT
   ';' .
63 constant_factor =  built_in_constant | constant_ref .
64 constant_id =  simple_id .
65 declaration =  function_decl | procedure_decl .
66 dependent_map_decl = DEPENDENT_MAP map_id AS target_parameter {
   target_parameter } [ map_subtype_of_clause ] dep_map_partition {
   dep_map_partition } END_DEPENDENT_MAP ';' .
67 dependent_view_decl = VIEW view_id ':' base_type ';' binding_header
   RETURN expression { binding_header RETURN expression } END_VIEW ';' .
68 dep_binding_decl = dep_from_clause [ where_clause ] [ ordered_by_clause
   ] .
69 dep_from_clause = FROM dep_source_parameter ';' { dep_source_parameter
   ';' } .
70 dep_map_decl_body =  dep_binding_decl map_project_clause .
71 dep_map_partition = [ PARTITION partition_id ':' ] dep_map_decl_body .
72 dep_source_parameter = source_parameter_id { ',' source_parameter_id }
   ':' ( simple_types | type_reference ) .
73 domain_rule =  [ label ':' ] logical_expression .
74 element =  expression [ ':' repetition ] .
75 entity_constructor =  entity_reference '(' [ expression
   { ',' expression } ] ')' .
76 entity_id =  simple_id .
77 entity_instantiation_loop =  FOR instantiation_loop_control ';'
   map_project_clause .
78 entity_reference =  [ ( source_schema_ref | target_schema_ref |
   schema_ref ) '.' ] entity_ref .
79 enumeration_reference =  [ type_reference '.' ] enumeration_ref .
80 escape_stmt =  ESCAPE ';' .
```



```
81  expression =  simple_expression [ rel_op_extended simple_expression ] .
82  expression_or_wild = expression | '_' .
83  extent_reference =  source_entity_reference | view_reference .
84  factor =  simple_factor [ '**' simple_factor ] .
85  foreach_expr =  EACH variable_id IN expression [ where_clause ] RETURN
    expression .
86  forloop_expr =  repeat_control RETURN expression .
87  formal_parameter =  parameter_id { ',' parameter_id } ':'
    parameter_type .
88  forward_path_qualifier = '::' attribute_ref [ path_condition ] .
89  for_expr =  FOR ( foreach_expr | forloop_expr ) .
90  from_clause =  FROM source_parameter ';' { source_parameter ';' } .
91  function_call =  ( built_in_function | function_ref ) [
    actual_parameter_list ] .
92  function_decl =  function_head [ algorithm_head ] stmt { stmt }
    END_FUNCTION ';' .
93  function_head =  FUNCTION function_id [ '(' formal_parameter { ';'
    formal_parameter } ')' ] ':' parameter_type ';' .
94  function_id =  simple_id .
95  generalized_types =  aggregate_type | general_aggregation_types |
    generic_type .
96  general_aggregation_types =  general_array_type | general_bag_type |
    general_list_type | general_set_type .
97  general_array_type =  ARRAY [ bound_spec ] OF [ OPTIONAL ] [ UNIQUE ]
    parameter_type .
98  general_attribute_qualifier = '.' ( attribute_ref | view_attribute_ref
    ) .
99  general_bag_type =  BAG [ bound_spec ] OF parameter_type .
100 general_list_type =  LIST [ bound_spec ] OF [ UNIQUE ] parameter_type .
101 general_ref =  parameter_ref | variable_ref | source_parameter_ref .
102 general_schema_alias_id = schema_id | schema_map_id | schema_view_id .
103 general_schema_ref = schema_ref | schema_map_ref | schema_view_ref .
104 general_set_type =  SET [ bound_spec ] OF parameter_type .
105 generic_type =  GENERIC [ ':' type_label ] .
106 group_qualifier = '\' entity_ref .
107 identified_by_clause =  IDENTIFIED_BY id_parameter ';' { id_parameter
    ';' } .
108 id_parameter = [ id_parameter_id ':' ] expression .
109 id_parameter_id = simple_id .
110 if_expr =  IF logical_expression THEN expression { ELSIF
    logical_expression expression } [ ELSE expression ] END_IF .
111 if_stmt =  IF logical_expression THEN stmt { stmt } [ ELSE stmt { stmt
    } ] END_IF ';' .
112 increment =  numeric_expression .
113 increment_control =  variable_id ':=' bound_1 TO bound_2
    [ BY increment ] .
114 index =  numeric_expression .
115 index_1 =  index .
116 index_2 =  index .
117 index_qualifier = '[' index_1 [ ':' index_2 ] ']' .
118 instantiation_foreach_control =  EACH variable_id IN expression { AND
    variable_id IN expression } [ INDEXING variable_id ] .
```



```
119 instantiation_loop_control =  instantiation_foreach_control |
    repeat_control .
120 integer_type =  INTEGER .
121 interval = '{' interval_low interval_op interval_item interval_op
    interval_high '}' .
122 interval_high =  simple_expression .
123 interval_item =  simple_expression .
124 interval_low =  simple_expression .
125 interval_op = '<' | '<=' .
126 label =  simple_id .
127 list_type =  LIST [ bound_spec ] OF [ UNIQUE ] base_type .
128 literal =  binary_literal | integer_literal | logical_literal |
    real_literal | string_literal .
129 local_decl =  LOCAL local_variable { local_variable } END_LOCAL ';' .
130 local_variable =  variable_id { ',' variable_id } ':' parameter_type [
    ':=' expression ] ';' .
131 logical_expression =  expression .
132 logical_literal =  FALSE | TRUE | UNKNOWN .
133 logical_type =  LOGICAL .
134 map_attribute_declaration =  [ target_parameter_ref [ index_qualifier ]
    [ group_qualifier ] '.' ] attribute_ref [ index_qualifier ] ':='
    expression ';' .
135 map_call =  [ target_parameter_ref '@' ] map_reference [
    partition_qualification ] '(' expression_or_wild { ','
    expression_or_wild } ')' .
136 map_decl =  MAP map_id AS target_parameter ';' { target_parameter ';' }
    ( map_subtype_of_clause subtype_binding_header map_decl_body ) | (
    binding_header map_decl_body { binding_header map_decl_body } ) END_MAP
    ';' .
137 map_decl_body =  ( entity_instantiation_loop {
    entity_instantiation_loop } )
    | map_project_clause
    | ( RETURN expression ';' ) .
138 map_id =  simple_id .
139 map_project_clause =  SELECT { map_attribute_declaration } .
140 map_reference =  [ schema_map_ref '.' ] map_ref .
141 map_subtype_of_clause =  SUBTYPE OF '(' map_reference ')' ';' .
142 multiplication_like_op = '*' | '/' | DIV | MOD | AND | '||' .
143 named_types =  entity_reference | type_reference | view_reference .
144 null_stmt = ';' .
145 number_type =  NUMBER .
146 numeric_expression =  simple_expression .
147 one_of =  ONEOF '(' supertype_expression
    { ',' supertype_expression } ')' .
148 ordered_by_clause =  ORDERED_BY expression { ',' expression } ';' .
149 parameter =  expression .
150 parameter_id =  simple_id .
151 parameter_type =  generalized_types | named_types | simple_types .
152 partition_id =  simple_id .
153 partition_qualification = '\' partition_ref .
154 path_condition = '{' extent_reference [ '|' logical_expression ] '}' .
155 path_qualifier = forward_path_qualifier | backward_path_qualifier .
156 population =  entity_reference .
```



```
157 precision_spec =  numeric_expression .
158 primary =  literal | ( qualifiable_factor { qualifier } ) .
159 procedure_call_stmt =  ( built_in_procedure | procedure_ref ) [
    actual_parameter_list ] ';' .
160 procedure_decl =  procedure_head [ algorithm_head ] { stmt }
    END_PROCEDURE ';' .
161 procedure_head =  PROCEDURE procedure_id [ '(' [ VAR ] formal_parameter
    { ';' [ VAR ] formal_parameter } ')' ] ';' .
162 procedure_id =  simple_id .
163 qualifiable_factor =  attribute_ref | constant_factor | function_call |
    general_ref | map_call | population | target_parameter_ref |
    view_attribute_ref | view_call .
164 qualifier =  general_attribute_qualifier | group_qualifier |
    index_qualifier | path_qualifier .
165 query_expression =  QUERY '(' variable_id '<*' aggregate_source '|'
    logical_expression ')' .
166 real_type =  REAL [ '(' precision_spec ')' ] .
167 reference_clause =  REFERENCE FROM schema_ref_or_rename [ '('
    resource_or_rename { ',' resource_or_rename } ')' ] [ AS ( SOURCE |
    TARGET ) ] ';' .
168 rel_op = '<' | '>' | '<=' | '>=' | '<>' | '=' | ':<>:' | ':=:' .
169 rel_op_extended =  rel_op | IN | LIKE .
170 rename_id =  constant_id | entity_id | function_id | procedure_id |
    type_id .
171 repeat_control =  [ increment_control ] [ while_control ] [
    until_control ] .
172 repeat_stmt =  REPEAT repeat_control ';' stmt { stmt } END_REPEAT ';' .
173 repetition =  numeric_expression .
174 resource_or_rename =  resource_ref [ AS rename_id ] .
175 resource_ref =  constant_ref | entity_ref | function_ref |
    procedure_ref | type_ref | view_ref | map_ref .
176 return_stmt =  RETURN [ '(' expression ')' ] ';' .
177 root_view_decl = VIEW view_id [ supertype_constraint ] ';'
    binding_header SELECT view_attr_decl_stmt_list { binding_header SELECT
    view_attr_decl_stmt_list } END_VIEW ';' .
178 rule_decl =  rule_head [ algorithm_head ] { stmt } where_clause
    END_RULE ';' .
179 rule_head =  RULE rule_id FOR '(' entity_ref { ',' entity_ref } ')'
    ';' .
180 rule_id =  simple_id .
181 schema_id =  simple_id .
182 schema_map_body_element =  function_decl | procedure_decl | view_decl |
    map_decl | dependent_map_decl | rule_decl .
183 schema_map_body_element_list =  schema_map_body_element {
    schema_map_body_element } .
184 schema_map_decl =  SCHEMA_MAP schema_map_id ';' reference_clause {
    reference_clause } [ constant_decl ] schema_map_body_element_list
    END_SCHEMA_MAP ';' .
185 schema_map_id =  simple_id .
186 schema_ref_or_rename =  [ general_schema_alias_id ':' ]
    general_schema_ref .
187 schema_view_body_element =  function_decl | procedure_decl | view_decl
    | rule_decl .
```



```
188 schema_view_body_element_list =  schema_view_body_element {
    schema_view_body_element } .
189 schema_view_decl =  SCHEMA_VIEW schema_view_id ';' { reference_clause }
    [ constant_decl ] schema_view_body_element_list END_SCHEMA_VIEW ';' .
190 schema_view_id =  simple_id .
191 selector =  expression .
192 set_type =  SET [ bound_spec ] OF base_type .
193 simple_expression =  term { add_like_op term } .
194 simple_factor =  aggregate_initializer | entity_constructor |
    enumeration_reference | interval | query_expression | ( [ unary_op ] (
    '(' expression ')' | primary ) ) | case_expr | for_expr | if_expr .
195 simple_types =  binary_type | boolean_type | integer_type |
    logical_type | number_type | real_type | string_type .
196 skip_stmt =  SKIP ';' .
197 source_entity_reference =  entity_reference .
198 source_parameter =  source_parameter_id ':' extent_reference .
199 source_parameter_id =  simple_id .
200 stmt =  assignment_stmt | case_stmt | compound_stmt | escape_stmt |
    if_stmt | null_stmt | procedure_call_stmt | repeat_stmt | return_stmt |
    skip_stmt .
201 string_literal =  simple_string_literal | encoded_string_literal .
202 string_type =  STRING [ width_spec ] .
203 subtype_binding_header = [ PARTITION partition_id ';' ] where_clause .
204 subtype_constraint =  OF '(' supertype_expression ')' .
205 subtype_declaration = SUBTYPE OF '(' view_ref { ',' view_ref } ')' .
206 subtype_view_decl = VIEW view_id subtype_declaration ';'
    subtype_binding_header SELECT view_attr_decl_stmt_list {
    subtype_binding_header SELECT view_attr_decl_stmt_list } END_VIEW ';' .
207 supertype_constraint = abstract_supertype_declaration |
    supertype_rule .
208 supertype_expression =  supertype_factor { ANDOR supertype_factor } .
209 supertype_factor =  supertype_term { AND supertype_term } .
210 supertype_rule =  SUPERTYPE [ subtype_constraint ] .
211 supertype_term =  view_ref | one_of | '(' supertype_expression ')' .
212 syntax_x =  schema_map_decl | schema_view_decl .
213 target_entity_reference =  entity_reference { '&' entity_reference } .
214 target_parameter =  target_parameter_id { ',' target_parameter_id } ':'
    [ AGGREGATE [ bound_spec ] OF ] target_entity_reference .
215 target_parameter_id =  simple_id ';' .
216 term =  factor { multiplication_like_op factor } .
217 type_id =  simple_id .
218 type_label =  type_label_id | type_label_ref .
219 type_label_id =  simple_id .
220 type_reference =  [ schema_ref '.' ] type_ref .
221 unary_op = '+' | '-' | NOT .
222 until_control =  UNTIL logical_expression .
223 variable_id =  simple_id .
224 view_attribute_decl =  view_attribute_id ':' [ OPTIONAL ] [
    source_schema_ref '.' ] base_type ':=' expression ';' .
225 view_attribute_id =  simple_id .
226 view_attr_decl_stmt_list = { view_attribute_decl } .
227 view_call =  view_reference [ partition_qualification ] '(' [
    expression_or_wild { ',' expression_or_wild } ] ')' .
```



```
228  view_decl =  ( root_view_decl | dependent_view_decl |
     subtype_view_decl ) .
229  view_id =  simple_id .
230  view_reference =  [ ( schema_map_ref | schema_view_ref ) '.' ]
     view_ref .
231  where_clause =  WHERE domain_rule ';' { domain_rule ';' } .
232  while_control =  WHILE logical_expression .
233  width =  numeric_expression .
234  width_spec = '(' width ')' [ FIXED ] .
```

B.3 Cross reference listing

