

INTERNATIONAL
STANDARD

ISO/IEC
12227

First edition
1995-02-15

**Information technology — Programming
languages — SQL/Ada Module Description
Language (SAMeDL)**

*Technologies de l'information — Langages de programmation — Langage
de description de modules SQL/Ada (SAMeDL)*



Reference number
ISO/IEC 12227:1995(E)

AVA Documents
Document Sharing Hub

Contents	Page
1 Scope	1
2 Normative references	1
3 Notations and Structures	2
3.1 Syntax Notation	2
3.2 Semantic Notation	2
3.3 Structure	3
3.4 Examples, Notes and Index Entries	3
4 Design Goals and Language Summary	3
4.1 Design Goals	3
4.2 Language Summary	4
4.2.1 Overview	4
4.2.2 Compilation Units	4
4.2.3 Modules	5
4.2.4 Procedures and Cursors	5
4.2.5 Domain and Base Domain Declarations	5
4.2.6 Other Declarations	6
4.2.7 Value Expressions and Typing	6
4.2.8 Standard Post Processing	6
4.2.9 Extensions	6
4.2.10 Default Values in Grammar	6
4.3 Entry Level SQL	7
5 Lexical Elements	7
5.1 Character Set	7
5.2 Lexical Elements, Separators, and Delimiters	7
5.3 Identifiers	8
5.4 Literals and Data Classes	9
5.5 Comments	10
5.6 Reserved Words	10

© ISO/IEC 1995

All rights reserved. Unless otherwise specified, no part of this publication may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying and microfilm, without permission in writing from the publisher.

ISO/IEC Copyright Office • Case Postale 56 • CH-1211 Genève 20 • Switzerland

Printed in Switzerland

6 Common Elements	10
6.1 Compilation Units	11
6.2 Context Clause	11
6.3 Table Names and the From Clause	12
6.4 References	12
6.5 Assignment Contexts and Conformance of an Expression to a Domain	17
6.6 Standard Post Processing	18
6.7 Extensions	18
7 Data Description Language and Data Semantics	19
7.1 Definitional Modules	19
7.1.1 Base Domain Declarations	20
7.1.1.1 Base Domain Parameters	20
7.1.1.2 Base Domain Patterns	21
7.1.1.3 Base Domain Options	22
7.1.2 The SAME Standard Base Domains	23
7.1.3 Domain and Subdomain Declarations	24
7.1.4 Constant Declarations	28
7.1.5 Record Declarations	29
7.1.6 Enumeration Declarations	31
7.1.7 Exception Declarations	32
7.1.8 Status Map Declarations	32
7.2 Schema Modules	34
7.2.1 Table Definitions	34
7.2.2 View Definitions	36
7.3 Data Conversions	37
8 Abstract Module Description Language	38
8.1 Abstract Modules	38
8.2 Procedures	39
8.3 Statements	43
8.4 Cursor Declarations	45
8.5 Cursor Procedures	49
8.6 Input Parameter Lists	52
8.7 Select Parameter Lists	54
8.8 Value Lists and Column Lists	56
8.9 Into_Clause and Insert_From_Clause	58
8.10 Value Expressions	60
8.11 Search Conditions	66
8.12 Subqueries	68
8.13 Status Clauses	68
9 Conformance	69
9.1 Introduction	69
9.2 Claims of Conformance	69
9.2.1 Introduction	69
9.2.2 Conformance via mapping.	69
9.2.2.1 Conformance via SQL module.	69
9.2.2.2 Conformance via embedded SQL Syntax.	69

9.2.3 Conformance via effects.	70
9.2.4 Multiple claims of conformance.	70
9.3 Extensions	70
Annex A. SAMeDL_Standard	71
Annex B. SAMeDL_System	84
Annex C. Standard Support Operations and Specifications	85
C.1 Standard Base Domain Operations	85
C.1.1 All Domains	85
C.1.2 Numeric Domains	86
C.1.3 Int and Smallint Domains	86
C.1.4 Character Domains	87
C.1.5 Enumeration Domains	87
C.1.6 Boolean Functions	88
C.1.7 Operations Available to the Application	88
C.2 Standard Support Package Specifications	90
C.2.1 SQL_Standard	90
C.2.2 SQL_Boolean_Pkg	91
C.2.3 SQL_Int_Pkg	92
C.2.4 SQL_Smallint_Pkg	95
C.2.5 SQL_Real_Pkg	98
C.2.6 SQL_Double_Precision_Pkg	101
C.2.7 SQL_Char_Pkg	104
C.2.8 SQL_Enumeration_Pkg	107
Annex D. Transform Chart	109
Annex E. Glossary	114
Annex F. References	117
Index	118

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work.

In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75 % of the national bodies casting a vote.

International Standard ISO/IEC 12227 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 22, *Programming languages, their environments and system software interfaces*.

Annexes A to C form an integral part of this International Standard. Annexes D to F are for information only.

Information technology - Programming languages - SQL/Ada Module Description Language (SAMeDL)

1 Scope

This International Standard specifies the syntax and semantics of a database programming language, the SQL Ada Module Description Language, SAMeDL. Texts written in the SAMeDL describe database interactions which are to be performed by database management systems (DBMS) implementing Database Language SQL. The interactions so described and so implemented are to be performed on behalf of application programs written in Programming Language Ada.

The SAMeDL is not a Programming Language; it may be used solely to specify application program database interactions and solely when those interactions are to occur between an Ada application program and an SQL DBMS.

The SAMeDL is defined with respect to Entry Level SQL. Therefore, all inclusions by reference of text from ISO/IEC 9075:1992 include all applicable Leveling Rules for Entry Level SQL.

This International Standard does not define the Programming Language Ada nor the Database Language SQL. Therefore, ISO 8652:1987 takes precedence in all matters dealing with the syntax and semantics of any Ada construct contained, referred or described within this International Standard; similarly, ISO/IEC 9075:1992 takes precedence in all matters dealing with the syntax and semantics of any SQL construct contained, referred or described within this International Standard.

Note: The SAMeDL is an example of an Abstract Modular Interface. A reference model for programming language interfaces to database management systems, which includes a description of Abstract Modular interfaces, can be found in reference [2].

2 Normative references

The following International Standards contain provisions which, through reference in this text, constitute provisions of this International Standard. At the time of publication, the editions indicated were valid. All International Standards are subject to revision, and parties to agreements based on this International Standard are encouraged to investigate the possibility of applying the most recent editions of the International Standards indicated below. Members of IEC and ISO maintain registers of currently valid International Standards.

ISO/IEC 9075:1992, *Information technology -- Database languages -- SQL*.

ISO/IEC 8652:1995, *Information technology -- Programming languages -- Ada*.

3 Notations and Structures

3.1 Syntax Notation

The context-free syntax of the language is described using a simple variant of the variant of BNF used in the description of Ada, (1.5 in ISO 8652:1987). The variation is:

- Underscores are preserved when using the name of a syntactic category outside of a syntax rule (1.5(6) of ISO 8652:1987).
- The italicized prefixes *Ada* and *SQL*, when appearing in the names of syntactic categories, indicate that an Ada or SQL syntactic category has been incorporated into this document. For example, the category *Ada_identifier* is identical to the category identifier as described in 2.3 of ISO 8652:1987; whereas the category *SQL_identifier* is identical to the category identifier as described in 5.4 of ISO/IEC 9075:1992.
- Numerical suffixes attached to the names of syntactic categories are used to distinguish appearances of the category within a rule or set of rules. An example of this usage is given below.

3.2 Semantic Notation

The meaning of a SAMeDL compilation unit (except where specified as implementation-defined) is given by:

- An Ada compilation unit, conforming to ISO 8652:1987.
- A module conforming to clause 12 of ISO/IEC 9075:1992.
- Interface rules, concerning the relationship between the SQL and Ada texts.

The semantics of SAMeDL constructs are given in part by collections of string transformers that produce Ada and SQL texts from SAMeDL input.

Note: A quick reference to these transformers appears in Annex D.

The effects of these transformers are described through the use of sample input strings. Those strings are written in a variant of the syntax notation. For example, the syntax of an *input_parameter* (see 8.6) is given by:

identifier_1 [named_phrase] : domain_reference [**not null**]

A representative input parameter declaration is given by

Id_1 [**named** *Id_2*] : *Id_3* [**not null**]

It is then possible to discuss the four variants of input parameters (the variants described by the presence or absence of optional phrases) in a single piece of text.

3.3 Structure

The remainder of this International Standard is structured in the following way. Clause 4 contains a descriptive overview of the goals and concepts of the SAMeDL. Clause 5 defines the lexical structure of the SAMeDL including the rules for identifier and literal formation and the list of reserved words. Clauses 6, 7 and 8 define the syntax and semantics of the SAMeDL. Each subclause of those clauses adheres to the following general format.

- The purpose of the item being defined by the subclause is introduced.
- The syntax of the item being defined is given in the notation described earlier.
- Abstract (non-context free) syntactical rules governing formation of instances of the item being defined are given, if applicable.
- The semantics of the item being defined are given. These semantics are given under the headings **Ada Semantics**, **SQL Semantics** and **Interface Semantics**, as appropriate.

3.4 Examples, Notes and Index Entries

Many of the subclauses of this International Standard are illustrated with examples. These examples are introduced by the words: “*Note: Examples*” on a line by themselves and are terminated by the words “*End Examples*,” likewise appearing on a line by themselves.

Note: Examples

This is an example of an example.

End Examples

Other notes also appear in this International Standard, introduced by the word *Note*. Both kinds of note are informative only and do not form part of the definition of the SAMeDL.

Items in the grammar that contain underscores are represented in the Index by a corresponding entry without underscores. For example, the “Ada identifier” entry in the index contains the page numbers of occurrences of both “Ada_identifier” and “Ada identifier”.

4 Design Goals and Language Summary

4.1 Design Goals

The SQL Ada Module Description Language (SAMeDL) is a Database Programming Language designed to automate the construction of software conformant to the SQL Ada Module Extensions (SAME) application architecture. This architecture is described in the document, *Guidelines for the Use of the SAME* [1].

The SAME is a *modular* architecture. It uses the concept of a Module as defined in 4.16 and 12 of ISO/IEC 9075:1992. As a consequence, a SAME-conformant Ada application does not contain embedded SQL statements and is not an embedded SQL Ada program as defined in 19.3 of ISO/IEC 9075:1992. Such a SAME-conformant application treats SQL in the manner in which Ada treats all other languages: it imports complete functional modules, not language fragments.

Modular architectures treat the interaction of the application program and the database as a design object. This results in a further isolation of the application program from details of the database design and implementation and improves the potential for increased specialization of software development staff.

Ada and SQL are vastly different languages: Ada is a Programming Language designed to express algorithms, while SQL is a Database Language designed to describe desired results. Text containing both Ada and SQL is therefore confusing and difficult to maintain. SAMeDL is a Database Programming Language designed to support the goals and exploit the capabilities of Ada with a language whose syntax and semantics is based firmly in SQL. Beyond modularity, the SAMeDL provides the application programmer the following services:

- An abstract treatment of null values. Using Ada typing facilities, a safe treatment of missing information based on SQL is introduced into Ada database programming. The treatment is safe in that it prevents an application from mistaking missing information (null values) for present information (non-null values).
- Robust status code processing. SAMeDL's Standard Post Processing provides a structured mechanism for the processing of SQL status parameters.
- Strong typing. SAMeDL's typing rules are based on the strong typing of Ada, not the permissive typing of SQL.
- Extensibility. The SAMeDL supports a class of user extensions. Further, it controls, but does not restrict, implementation defined extensions.

4.2 Language Summary

4.2.1 Overview

The SAMeDL is designed to facilitate the construction of Ada database applications that conform to the SAME architecture as described in [1]. The SAME method involves the use of an abstract interface, an abstract module, a concrete interface, and a concrete module. The abstract interface is a set of Ada package specifications containing the type and procedure declarations to be used by the Ada application program. The abstract module is a set of bodies for the abstract interface. These bodies are responsible for invoking the routines of the concrete interface, and converting between the Ada and the SQL data and error representations. The concrete interface is a set of Ada specifications that define the SQL procedures needed by the abstract module. The concrete module is a set of SQL procedures that implement the concrete interface.

Within this International Standard, the concrete module of [1] is called an SQL module and its contents are given under the headings **SQL Semantics** within the clauses of this specification. The abstract modules of [1] are given under the heading **Ada Semantics** within the clauses of this specification.

4.2.2 Compilation Units

A *compilation unit* consists of one or more modules. A module may be either a definitional module containing shared definitions, a schema module containing table, view, and privilege definitions, or an abstract module containing local definitions and procedure and cursor declarations.

4 Database Programming Language - SAMeDL

4.2.3 Modules

A *definitional module* contains the definitions of base domains, domains, constants, records, enumerations, exceptions, and status maps. Definitions in definitional modules may be seen by other modules.

A *schema module* contains the definitions of tables, views, and privileges.

An *abstract module* defines (a portion of) an application's interface to the database: it defines SQL services needed by an Ada application program. An abstract module may contain procedure declarations, cursor declarations, and definitions such as those that may appear in a definitional module. Definitions in an abstract module, however, may not be seen by other modules.

4.2.4 Procedures and Cursors

A *procedure* declaration defines a basic database operation. The declaration defines an Ada procedure declaration and a corresponding SQL procedure. A SAMeDL procedure consists of a single statement along with an optional input parameter list and an optional status clause. The input parameter list provides the mechanism for passing information to the database at runtime. A statement in a SAMeDL procedure may be a commit statement, rollback statement, insert statement query, insert statement values, update statement, select statement or an implementation-defined extended statement. The semantics of a SAMeDL statement directly parallel that of its corresponding SQL statement.

SAMeDL *cursor* declarations directly parallel SQL cursor declarations. In contrast to the language of ISO/IEC 9075:1992, the procedures that operate on cursors, procedures containing either an open, fetch, close, update positioned or delete positioned statement, are packaged with the declaration of the cursor upon which they operate, thereby improving readability. Further, if no procedure containing an open, fetch or close statement is explicitly given in a cursor declaration, the language provides such procedures implicitly, thereby improving writeability (ease of use).

4.2.5 Domain and Base Domain Declarations

Objects in the language have an associated *domain*, which characterizes the set of values and applicable operations for that object. In this sense, a domain is similar to an Ada type.

A *base domain* is a template for defining domains. A base domain declaration consists of a set of parameters, a set of patterns and a set of options. The parameters are used to supply information needed to declare a domain or subdomain derived from the base domain. Patterns contain templates for the generation of Ada code to support the domain in Ada applications. This code generally contains type declarations and package instantiations. Options contain information needed by the compiler. Parameters may be used in the patterns and options and their values may be referenced in other statements.

Base domains are classified according to their associated data class. A data class is either integer, fixed, float, enumeration, or character. A numeric base domain has a data class of either integer, fixed, or float. An enumeration base domain has a data class of enumeration, and defines both an ordered set of distinct enumeration literals and a bijection between the enumeration literals and their associated database values. A character base domain has a data class of character.

4.2.6 Other Declarations

Certain SAMeDL declarations are provided as a convenience for the user. For example, *constant* declarations name and associate a domain with a static expression. *Record* declarations allow distinct procedures to share types. An *exception* declaration defines an Ada exception declaration with the same name.

4.2.7 Value Expressions and Typing

Value expressions are formed and evaluated according to the rules of SQL, with the exception that the strong typing rules are based on those of Ada. In the typing rules of the SAMeDL, the domain acts as an Ada type in a system without user defined operations. Strong typing necessitates the introduction of domain conversions. These conversions are modeled after Ada type conversions; the operational semantics of the SAMeDL domain conversion is the null operation or identity mapping. The language rules specify that an informational message be displayed under circumstances in which this departure from the Ada model has visible effect.

4.2.8 Standard Post Processing

Standard post processing is performed after the execution of an SQL procedure but before control is returned to the calling application procedure. The *status clause* from a SAMeDL procedure declaration attaches a *status mapping* to the application procedure. That status mapping is used to process SQL status data in a uniform way for all procedures and to present SQL status codes to the application in an application-defined manner, either as a value of an enumerated type, or as a user defined exception. SQL status codes not specified by the status map result in a call to a standard database error processing procedure and the raising of the predefined SAMeDL exception, `SQL_Database_Error`. This prevents a database error from being ignored by the application.

4.2.9 Extensions

The data semantics of the SAMeDL may be extended without modification to the language by the addition of user-defined base domains. For example, a user-defined base domain of `DATE` may be included without modification to the SAMeDL.

DBMS specific (i.e., non-standard) operations and features that require compiler modification (e.g., dynamic SQL) may also be included into the SAMeDL. Such additions to the SAMeDL are referred to as extensions. Schema elements, table elements, statements, query expressions, query specifications, and cursor statements may be extended. The modules, tables, views, cursors, and procedures that contain these extensions are marked (with the keyword **extended**) to indicate that they go outside the standard.

4.2.10 Default Values in Grammar

Obvious but overridable defaults are provided in the grammar. For example, open, close, and fetch statements are essential for a cursor, but their form may be deduced from the cursor declaration. The SAMeDL will therefore supply the needed open, close, and fetch procedure declarations if they are not supplied by the user.

4.3 Entry Level SQL

Within the text of this specification, the name SQL references the language known as Entry Level SQL in ISO/IEC 9075:1992. Features and capabilities of ISO/IEC 9075:1992 that do not lie within Entry Level SQL may be implemented via the extension facility defined in this International Standard. See 6.7 of this specification.

5 Lexical Elements

The text of a compilation is a sequence of lexical elements, each composed of characters from the basic character set. The rules of composition are given in this chapter.

5.1 Character Set

The only characters allowed in the text of a compilation are the basic characters and the characters that make up character literals (described in 5.4 of this specification). Each character in the basic character set is represented by a graphical symbol.

```
basic_character ::=
    upper_case_letter | lower_case_letter | digit |
    special_character | space_character
```

The characters included in each of the above categories of the basic characters are defined as follows:

1. upper_case_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
2. lower_case_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
3. digit
0 1 2 3 4 5 6 7 8 9
4. special_character
' () * + , - . / : ; < = > _ |
5. space_character

5.2 Lexical Elements, Separators, and Delimiters

The text of each compilation is a sequence of separate lexical elements. Each lexical element is either an identifier (which may be a reserved word), a literal, a comment, or a delimiter. The effect of a compilation depends only on the particular sequences of lexical elements excluding the comments, if any, as described in this chapter. Identifiers, literals, and comments are discussed in the following clauses. The remainder of this clause discusses delimiters and separators.

An explicit separator is required to separate adjacent lexical elements when, without separation, interpretation as a single lexical element is possible. A separator is any of a space character, a format effector, or the end of a

line. A space character is a separator except within a comment or a character literal. Format effectors other than horizontal tabulation are always separators. Horizontal tabulation is a separator except within a comment.

The end of a line is always a separator. The language does not define what causes the end of a line.

One or more separators are allowed between any two adjacent lexical elements, before the first lexical element of each compilation, or after the last lexical element of each compilation. At least one separator is required between an identifier or a numeric literal and an adjacent identifier or numeric literal.

A delimiter is one of the following special characters

```
( ) * + , - . / : ; < = > |
```

or one of the following compound delimiters each composed of two adjacent special characters

```
=> .. := <> >= <=
```

Each of the special characters listed for single character delimiters is a single delimiter except if that character is used as a character of a compound delimiter, a comment, or a literal.

Each lexical element shall fit on one line, since the end of a line is a separator. The single quote and underscore characters, as well as two adjacent hyphens, are not delimiters, but may form part of other lexical elements.

5.3 Identifiers

```
identifier ::= SQL_actual_identifier
```

The length restrictions that apply to *SQL_identifiers* (see 5.2, syntax rules 8, 9 and 5.5 syntax rule 3 of ISO/IEC 9075:1992) *do not apply* to SAMeDL identifiers. Whenever two identifiers are deemed equivalent, it is in the sense of the rules of SQL (see 5.2, syntax rules 10 through 14 of ISO/IEC 9075:1992).

Note. An *SQL_actual_identifier* is either a *delimited_identifier* or a *regular_identifier*. The form of an SQL *regular_identifier* is essentially the same as the Ada identifier, except that a *regular_identifier* may end in an underscore. A *delimited_identifier* is *any* character string within a pair of doublequote characters ("). *Delimited identifiers* provide a means of using tokens that would otherwise be reserved words (see 5.6 of this specification) as identifiers. Thus **fetch** is a reserved word, but the construct

```
procedure "fetch" is fetch;
```

defines a procedure named "fetch" that contains a fetch statement.

Identifier equivalence in SQL is similar to Ada identifier equivalence for regular identifiers. Equivalence for delimited identifiers is case sensitive. Equivalence of regular identifiers with delimited identifiers proceeds by considering the regular identifier to be all upper case and then doing a case sensitive comparison to the delimited identifier. So a column named *Status* is not identified by the delimited identifier "Status" but is identified by the delimited identifier "STATUS". *end Note*

Ada Semantics

Let *ident* be an identifier. Define *AdaID(ident)* by

```
AdaID(ident) = ident if ident is a regular identifier
              id    if ident is the delimited identifier "id"
```

Note: If *ident* is an identifier, *AdaID(ident)* is not necessarily an *Ada_identifier*.

8 Database Programming Language - SAMeDL

SQL Semantics

Let SQL_NAME be a function on identifiers into the set of $SQL_identifiers$ with the property that $SQL_NAME(Id_1)$ and $SQL_NAME(Id_2)$ shall be equivalent if and only if Id_1 and Id_2 are equivalent (see above).

Note: Entry level SQL identifiers are limited to eighteen characters in length (see 5.2, leveling rule 2.a of ISO/IEC 9075:1992) but SAMeDL identifiers are not. SQL_NAME deals with this discrepancy. SQL_NAME is not applied to externally visible names, i.e., schema, table, view and column names, but only to names within SQL modules, e.g., cursor, procedure and parameter names.

5.4 Literals and Data Classes

Literals follow the SQL literal syntax (5.3 of ISO/IEC 9075:1992). There are five classes into which literals are placed based on their lexical properties: **character**, **integer**, **fixed**, **float**, and **enumeration**.

```

literal ::=
    database_literal | enumeration_literal

database_literal ::=
    character_literal | [+ | -] numeric_literal

numeric_literal ::=
    integer_literal | fixed_literal | float_literal

character_literal ::=
    ' {character} '

character ::=
    implementation defined

integer_literal ::=
    digit {digit}

fixed_literal ::=
    integer_literal . integer_literal |
    . integer_literal |
    integer_literal .

float_literal ::=
    fixed_literal exp [+ | -] integer_literal

exp ::=
    e | E

enumeration_literal ::= identifier

```

1. Each literal has an associated data class, denoted $DATACLASS(L)$. In particular:

if L is a character_literal then $DATACLASS(L)$ is	character
integer_literal	integer
fixed_literal	fixed
float_literal	float
enumeration_literal	enumeration

(see data_class in 7.1.1.1 of this specification).

2. If Id is an enumeration literal, then $AdaID(id)$ shall be an *Ada_identifier*. In particular, an enumeration literal shall not be an Ada character literal.

3. Every character_literal *CL* has an associated length $\text{LENGTH}(CL)$ in the sense of 5.3, Syntax Rule 5 in ISO/IEC 9075:1992. For any non character literal, *L*, $\text{LENGTH}(L)=\text{NO_LENGTH}$.
4. Integer, fixed and float literals are collectively known as numeric literals. Every numeric literal *NL* has a scale, $\text{SCALE}(NL)$. An integer literal has scale 0. The scale of a float literal is equal to the scale of any other float literal and larger than the scale of any non-float numeric literal. The scale of a fixed literal is the number of digits appearing to the right of the decimal point within the literal. Any non numeric literal, *L*, has $\text{SCALE}(L)=\text{NO_SCALE}$. See 6.4 of this specification for the interpretation of enumeration literals.
5. The single quote or “tic” character can be included in a character_literal by duplication in the usual way. Thus the string of tics: ' ' ' ' represents a character_literal of length one containing the single quote as its only character.

5.5 Comments

A comment starts with two adjacent hyphens and extends up to the end of the line. A comment can appear on any line of a module. The presence or absence of comments has no influence on whether a module is legal or illegal. Furthermore, comments do not influence the meaning of a module; their sole purpose is the enlightenment of the human reader.

5.6 Reserved Words

The following is the list of the SAMeDL reserved words:

ABSTRACT	ALL	AND	ANY	AS	ASC
AUTHORIZATION	AVG	BASE	BETWEEN	BODY	BY
CHECK	CLASS	CLOSE	COMMIT	CONSTANT	CONVERSION
COUNT	CURRENT	CURSOR	DATA	DBMS	DECLARE
DEFAULT	DEFINITION	DELETE	DERIVED	DESC	DISTINCT
DOMAIN	END	ENUMERATION	ESCAPE	EXISTS	EXCEPTION
EXTENDED	FETCH	FOR	FOREIGN	FROM	GRANT
GROUP	HAVING	IMAGE	IN	INSERT	INTO
IS	KEY	LIKE	MAP	MAX	MIN
MODULE	NAME	NAMED	NEW	NOT	NULL
OF	ON	OPEN	OPTION	OR	ORDER
OUT	PATTERN	POS	PRIMARY	PRIVILEGES	PROCEDURE
PUBLIC	RAISE	RECORD	REFERENCES	ROLLBACK	SCALE
SCHEMA	SELECT	SET	SOME	STATUS	SUBDOMAIN
SUM	TABLE	TO	TYPE	UNION	UNIQUE
UPDATE	USE	USER	USES	VALUES	VIEW
WHERE	WITH	WORK			

6 Common Elements

6.1 Compilation Units

A *compilation unit* is the smallest syntactic object that can be successfully compiled. It consists of a sequence of one or more modules.

```

compilation_unit ::=
    module {module}

module ::=
    definitional_module | abstract_module | schema_module

```

6.2 Context Clause

The context clause is a means by which a module gains visibility to names defined in other modules. The syntax and semantics of context clauses are similar to the syntax and semantics of Ada context clauses (8.4, 10.1.1 of ISO 8652:1987) but there are differences.

```

context ::=
    context_clause {context_clause}

context_clause ::=
    with_clause | use_clause | with_schema_clause

with_clause ::=
    with module_name [as_phrase]
        {, module_name [as_phrase]} ;

use_clause ::=
    use module_name {, module_name} ;

with_schema_clause ::=
    with schema schema_name [as_phrase]
        {, schema_name [as_phrase]} ;

module_name ::=
    identifier

schema_name ::=
    SQL_schema_authorization_identifier

as_phrase ::=
    as identifier

```

1. Consider the following with_clause and with_schema_clause:

```

        with M [as N1];
        with schema S [as N2];

```

In these clauses, M shall be the name of a definitional module and S the name of a schema module. The name M of the definitional module is said to be *exposed* if the as_phrase is not present in the context_clause; otherwise the name M is *hidden* and the name N₁ is the exposed name of M. Similar comments apply to S and N₂. The name of a module (see 7.1, 7.2, and 8.1 of this specification) is its exposed name within the text of that module. Within the text of any module, no two exposed module names shall be the same.

2. A module_name in a use_clause shall be the exposed name of a definitional_module that is an operand of a prior with_clause.

3. The scope of a `with_clause` or `use_clause` in the context of a module (see 7.1, 7.2, and 8.1 of this specification) is the text of that module.

4. Only an abstract or schema module context may contain a `with_schema_clause`.

Note: As a consequence of these definitions, abstract modules cannot be brought into the context of (**withed** by) another module.

6.3 Table Names and the From Clause

The table names in insert, update, and delete statements and the from clauses of select statements, cursor declarations, and subqueries (see 8.3, 8.4 and 8.12 of this specification) also make names, in particular column names, visible. The `from_clause` differs from an *SQL* `from_clause` (6.3 and 7.4 of ISO/IEC 9075:1992) only in the optional appearance of the **as** keyword, which is inserted for uniformity with the remainder of the language.

```

from_clause ::=
    from table_ref {, table_ref}

table_ref ::=
    table_name [[as] SQL_correlation_name]

table_name ::=
    [schema_ref.]identifier

schema_ref ::=
    schema_name | identifier

```

1. If present, `schema_ref` shall be either the `schema_name` in the authorization clause of the abstract module in which the `table_name` appears (8.1 of this specification) or the exposed name of a schema module in the context of the module in which `table_name` appears (6.2 of this specification). In either case, the identifier shall be the name of a table or the name of a view within that schema module. If the `schema_ref` is absent from the `table_name`, then the identifier shall be the name of a table or the name of a view within the schema module named in the authorization clause of the module in which the `table_name` appears.
2. If the correlation name is not present in a `table_ref`, then the table name or view name in the `table_ref` is *exposed*; otherwise the table name or view name is hidden and the `correlation_name` is exposed. No two exposed names within a `from_clause` shall be the same.
3. For the scope of table and view names see Syntax Rule 2 of 6.3 in ISO/IEC 9075:1992.

6.4 References

The rules concerning the meaning of references are modeled on those of Ada and those of SQL. As neither module nesting nor program name overloading occurs, these rules are fairly simple, and are therefore listed. For the purposes of this clause, an *item* is either:

- A definitional module (See 7.1), an abstract module (8.1), or a schema module (7.2 of this specification).
- A procedure (See 8.2), a cursor (8.4), or a procedure within a cursor (8.5 of this specification).
- Anything in the syntactic category 'definition' as given by 7.1 of this specification. This includes base domains, domains, subdomains, enumerations, constants, records, and status maps.

- A domain parameter (See 7.1.1.1 of this specification.)
- An enumeration literal within an enumeration (See 5.4 and 7.1.6 of this specification).
- An exception (7.1.7 of this specification).
- An input_parameter of a procedure or cursor declaration (See 8.2, 8.4, and 8.6 of this specification).
- A table defined within a schema module (See 7.2 of this specification).
- A column defined within a table (See 7.2 of this specification).

A location within the text of a module is said to be a *defining* location if it is the place of

- The name of an item within the item's declaration. (*Note:* This includes enumeration literals within the declaration of an enumeration and domain parameters within the declaration of a domain.)
- The name of a table in a from_clause.
- The name of the target table of an insert, update, or delete statement.
- A schema_name or module_name in a context_clause.

Text locations not within comments that are not defining locations are *reference* locations. An identifier that appears at a reference location is a reference to an item. The meaning of that reference in that location, that is, the identity of the item referenced, is defined by the rules of this clause. When these rules determine more than one meaning for an identifier, then all items referenced shall be enumeration literals.

```

module_reference ::= identifier

schema_reference ::= schema_name | identifier

base_domain_reference ::= [module_reference.]identifier

domain_reference ::= [module_reference.]identifier

domain_parameter_reference ::= domain_reference.identifier

subdomain_reference ::= [module_reference.]identifier

enumeration_reference ::= [module_reference.]identifier

enumeration_literal_reference ::= [module_reference.]identifier

exception_reference ::= [module_reference.]identifier

constant_reference ::= [module_reference.]identifier

record_reference ::= [module_reference.]identifier

procedure_reference ::= [module_reference.]identifier

cursor_reference ::= [module_reference.]identifier

cursor_proc_reference ::= [cursor_reference.]identifier

```

```

input_reference ::= [procedure_reference.]identifier
                  | [cursor_proc_reference.]identifier

status_reference ::= [module_reference.]identifier

table_reference ::= [schema_reference.]SQL_identifier

column_name ::= SQL_identifier

column_reference ::= [table_reference.]column_name
                   (See 6.4 of ISO/IEC 9075:1992).

```

A reference is a simple name (an identifier) optionally preceded by a prefix: a sequence of as many as three identifiers, separated by dots.

For the purposes of this clause, the “text of a cursor” does not include the text of the procedures, if any, contained in the cursor. A dereferencing rule is said to “determine a denotation” for a reference if it either (i) specifies an item to which the reference refers, or (ii) determines that the reference is not valid.

Note: Unlike the Ada dereferencing rules (see 8.2 and 8.3 in ISO 8652:1987), the SAMeDL rules treat the prefix as a whole, not component by component.

Prefix Denotations

The prefix of a reference shall denote one of the following:

- An abstract module, procedure, cursor or cursor procedure, but only from within the text of the abstract module, procedure, cursor, or cursor procedure.
- A table, if the table is in scope at the location in which the reference appears.
- A domain.
- A definitional or schema module.

Note: As a consequence of the rule given earlier, that all meanings of an identifier with multiple meanings must be enumeration literals, a prefix may have at most one denotation or meaning, as it may not denote an enumeration literal.

Let L be the reference location of prefix P . Let X , Y , and Z be simple names. Then

1. If L is within the text of a cursor procedure U , then P denotes
 - a. The cursor procedure U if either
 - i. P is of the form X and X is the simple name of U ; or
 - ii. P is of the form $X.Y$; X is the name of the cursor containing L (and therefore also U); in which case Y shall be the simple name of U else the prefix is not valid; or
 - iii. P is of the form $X.Y.Z$; X is the name of the module containing L ; Y is the name of the cursor containing L (and therefore also U); in which case Z shall be the simple name of U else the prefix is not valid;
 - b. The table T being updated in a cursor_update_statement, if the statement within the cursor procedure containing L is a cursor_update_statement and either

- i. P is of the form X and X is the simple name of T ; or
 - ii. P is of the form $X.Y$; X is an exposed name for the schema module S containing the declaration of T and Y is the simple name of T .
- 2. If rule 1 does not determine a denotation for P , then P denotes
 - a. The cursor or procedure R , if L is within the text of R and either
 - i. P is of the form X and X is the simple name of R ; or
 - ii. P is of the form $X.Y$; X is the name of the module containing L (and therefore also R); Y is the simple name of R ;
 - b. The table T , if L is in the scope of table name T (Rule 3 in 6.3 of this specification), and either
 - i. P is of the form X and X is a simple name exposed for T ; or
 - ii. P is of the form $X.Y$; X is the exposed name of the schema module containing the table T and Y is a simple name of T .
- 3. If rules 1 and 2 do not determine a denotation for P , then P denotes the domain R ,
 - a. If P is of the form X and X is the simple name of R and the declaration of R appears in the module containing L and precedes L within that module; or
 - b. P is of the form $X.Y$; X is the exposed name of the module containing the declaration of R and Y is the simple name of R .
- 4. If none of the above rules determines a denotation for P , then P is a simple name that denotes the
 - a. Definitional module M if either
 - i. L is in the scope of a `with_clause` exposing P as the name of M ; or
 - ii. L is in the definitional module M and P is the name of M .
 - b. Schema module S if either
 - i. L is in the scope of a `with_schema_clause` exposing P as the name of S ; or
 - ii. L is in an abstract module whose authorization clause identifies S and P is the name of S ;
 - c. Abstract module M if L is within the text of M and P is the name of M ;
 - d. Domain D , if D is declared within a module N such that there is a `use clause` for N in the module containing L , and P is the name of D .

Denotations of Full Names

Let L be the location of a reference ld . Then ld is a reference to the item lm if lm is not a module, procedure, cursor or cursor procedure, or table and

1. ld is of the form $P.X$ where X is the name of lm and P is a prefix denoting
 - a. A definitional module containing the declaration of lm ;
 - b. The abstract module, M , in which L appears, and lm is declared in M at a text location that precedes L ;
 - c. The procedure, cursor, or cursor procedure that contains L , and lm is an input parameter to that procedure, cursor, or cursor procedure;
 - d. A table, in which case lm is a column within that table;
 - e. A schema module, in which case lm is a table within that module.
 - f. A domain, in which case lm is a parameter in that domain.
2. ld is of the form X and X is the name of lm . Then
 - a. L appears in a cursor, procedure, or cursor procedure (See 8.4, 8.2, and 8.5 of this specification) and
 - i. lm is an input parameter to that cursor, procedure, or cursor procedure;
 - ii. lm is a column of one of the tables in scope of L ;
 - b. If rule (a) does not determine a denotation for ld , then lm is declared in the module containing L at a location preceding L ;
 - c. If neither rule (a) nor (b) determines a denotation for ld , then lm is declared within a module M such that the module containing L has a use clause for M .

Note: An item lm is *visible* at location L if there exists a name ld (either simple or preceded by a prefix) such that if ld were at location L , then ld would be a reference to lm .

Note: Examples

The following clause contains examples of the disambiguation of prefixes, with the applicable rule in a comment. At the point in Proc1 marked by Note1, the declaration of constant Inp2 is hidden by the input parameter Inp2: that constant would have to be qualified by the prefix "Abmod" to be visible. At the point in Proc2 marked by Note2, if COL is the name of a column of visible table TABNAME, then that reference is ambiguous. A reference to the input parameter would have to be "Proc2.Col", while a reference to the table column would have to be "TABNAME.COL". Finally, Dom1 is visible at Note3 since Defmod is in both a **with** and **use** clause in Abmod. Without the **use** clause, the reference at Note3 would have to be to "Defmod.Dom1".

```
with SAMeDL_Standard;
use SAMeDL_Standard;
definition module Defmod is
  constant Newfirst is 0;
  constant Newlast is 999;
  domain Dom1 is new SQL_Int(First => Newfirst,
                               Last => Defmod.Newlast); -- 4a(ii)
end Defmod;

with Defmod, SAMeDL_Standard;
use Defmod, SAMeDL_Standard;
with schema Sname1;
```

```

abstract module Abmod is
  authorization Sname2
  constant Newfirst is 0;
  constant Inp2      : Dom1 is 1; -- Note3
  domain Dom is new SQL_Int (First => Abmod.Newfirst, -- 4c
                              Last => Defmod.Newlast); -- 4a(i)
  procedure Proc (Inp1 : Dom; Inp2 : Dom) is
    insert into TABNAME
      select Proc.Inp1,      -- 2a(i)
             Abmod.Proc.Inp2 -- 2a(ii)
      from TABNAME
  ;
  cursor Curse
  for
    select COL
    from Sname1.TAB -- 4b(i)
  ;
  is
    procedure Proc1 (Inp1 : Dom; Inp2 : Dom; Inp3 : Dom) is
      update Sname2.TABNAME      -- 4b(ii)
      set COL1 = Proc1.Inp1,      -- 1a(i)
          COL2 = Curse.Proc1.Inp2, -- 1a(ii)
          COL3 = Abmod.Curse.Proc1.Inp3, -- 1a(iii)
          TABNAME.COL4 = Inp1,      -- 2b(i)
          Sname2.TABNAME.COL5 = Inp2 -- 2b(ii) : Note1
    ;
  end Curse;
  procedure Proc2 (Col : Dom) is
    insert into TABNAME
      select COL      -- Note2
      from TABNAME
  ;
  cursor Curset1
  for
    select COL
    from Sname2.TABNAME -- 4b(ii)
  ;
end Abmod;

```

End Examples

6.5 Assignment Contexts and Conformance of an Expression to a Domain

A value expression (see 8.10 of this specification) is said to appear in an *assignment context* if it is either

- The static expression in a constant declaration (see 7.1.4 of this specification),
- A select parameter (see 8.7 of this specification),
- A value in an insert_value_list (see 8.8 of this specification), or
- The right hand side of a set_item within an update_statement (see 8.3 of this specification).

A value expression *VE* is said to *conform* to a domain *D* under the following conditions.

- If DOMAIN(*VE*) ≠ NO_DOMAIN, then DOMAIN(*VE*) = *D*.
- If DATACLASS(*D*) is **integer** or **fixed**, then DATACLASS(*VE*) is **integer** or **fixed**.
- If DATACLASS(*D*) is **float**, then DATACLASS(*VE*) is **integer**, **fixed**, or **float**.
- If DATACLASS(*D*) is **character**, then DATACLASS(*VE*) is **character**.

-- If DATACLASS(*D*) is **enumeration**, then if DOMAIN(*VE*) = NO_DOMAIN, then *VE* is an enumeration literal in *D*.

6.6 Standard Post Processing

Standard post processing is the processing that is done after execution of an SQL procedure, but before control is returned to the calling application. That processing is described as follows:

1. The SQL status parameter in an SQL procedure call is SQLSTATE unless a status clause appears in the procedure declaration that references a status map specifying **sqlcode**, in which case the SQL status parameter is SQLCODE. See 7.1.8 and 8.13 of this specification.

Note: SQLSTATE is the preferred status parameter. SQLCODE is a deprecated feature of ISO/IEC 9075:1992.

2. If a status map is attached to the procedure via a status clause (see 8.13 of this specification), then if the value of the status parameter appears in the left hand side of some status_assignment in that status map (possibly via the equivalences in 7.1.8 of this specification) then the Ada procedure's status parameter is set to the value of the right hand side of that status_assignment, if that right hand side is an enumeration_literal; if that right hand side is a **raise** statement, then the named *Ada_exception* is raised. This is not considered an error condition in the sense of the next paragraph. In particular, SQL_Database_Error_Pkg.Process_Database_Error is not called.

3. If the value of the SQL status parameter does not appear in the left hand side of any status_assignment in the map attached to the procedure or there is no status map attached to the procedure and the SQL status parameter indicates a condition other than successful completion, then an error condition exists. In this case the parameterless procedure SQL_Database_Error_Pkg.Process_Database_Error is called. Upon return from that procedure, the exception SAMeDL_Standard.SQL_Database_Error is raised.

Note: Successful completion is indicated by an SQLSTATE value of "00000" or an SQLCODE value of 0. In particular, warning values, SQLSTATE values "01xxx" or positive SQLSTATE values other than 100, are not considered to indicate successful completion in the sense of this paragraph.

6.7 Extensions

Extended tables, views, modules, procedures, and cursors allow for the inclusion into the SAMeDL of DBMS-specific, that is, non-standard, operations and features, while preserving the benefits of standardization. These DBMS-specific extensions may be *verbs*, such as connect and disconnect, that signal the beginning and end of program execution, or *functions*, such as date manipulation routines, that extract the month from a date. The use of extensions, particularly the **extended** keyword, serves to mark those modules, tables, views, cursors, and procedures that go outside the standard and may require effort should the underlying DBMS be changed.

```
extended_schema_element ::=
    implementation defined

extended_table_element ::=
    implementation defined

extended_statement ::=
    implementation defined

extended_query_expression ::=
    implementation defined

extended_query_specification ::=
    implementation defined
```

18 Database Programming Language - SAMeDL

```
extended_cursor_statement ::=
    implementation defined
```

Notice that the grammar is arranged such that extensions cannot influence the formation of

- procedure names
- cursor names
- module names
- table names
- view names
- status clauses, status parameter names and types, and standard post processing

Although the syntax and semantics of extensions are implementation defined, any portion of an extension whose semantics is expressible in standard SAMeDL shall be expressed in standard SAMeDL syntax. An extension may expand the class of:

- Value expressions by adding operators and functions. The operands of those operators and functions shall be restricted by rules similar to those in 6.5 and 8.10 of this specification.
- Search conditions by adding atomic predicates. Operands of those atomic predicates shall likewise be restricted according to rules such as those in 8.11 of this specification.
- Input parameter lists by adding the mode **out** to the parameter declarations, according to the rules of 8.6 of this specification.
- Table elements. If the extended table element is in the form of a column definition, the domain reference shall be present (see 7.2.1 of this specification).

Similarly, database data returned from extended procedures and cursors shall be defined in the syntax and semantics of select parameter lists (8.7 of this specification) with the syntax and semantics of the extended value expression class replacing the standard syntax and semantics. Such outputs shall be record objects. An extended statement returning such data shall accept an `into_clause` as described in 8.9 of this specification for specifying the record parameter name and type.

7 Data Description Language and Data Semantics

7.1 Definitional Modules

Definitional modules contain declarations of base domains, domains, subdomains, constants, records, enumerations, exceptions, and status maps. An Ada library unit package declaration is defined for each definitional module.

```
definitional_module ::=
```

```

[context]
[extended] definition module identifier_1 is
    {definition}
end [identifier_2];

definition ::=
    base_domain_declaration |
    domain_declaration | subdomain_declaration |
    constant_declaration | record_declaration |
    enumeration_declaration | exception_declaration |
    status_map_declaration

```

1. When present, identifier_2 shall be equivalent to identifier_1. (See 5.3 of this specification.)

Notes:

- No with_schema_clause shall appear in the context of a definitional module. (See 6.2 of this specification.)
- No two declarations within a definitional module shall have the same name, except for enumeration literals (see 7.1.6 of this specification).

End Notes.

Ada Semantics

For each definitional module within a compilation unit there is a corresponding Ada library unit package the name of which is the name of the definitional module, that is, AdaID(identifier_1). The Ada construct giving the Ada semantics of each definition within a definitional module is declared within the specification of that package. Nothing else appears in the specification of that package.

7.1.1 Base Domain Declarations

Base domains are the basis on which domains are defined. A base domain declaration has three parts: a sequence of parameters, used in domain declarations to supply information to the other two parts; a sequence of patterns, used to produce Ada source code in support of a domain; and a sequence of options, used by the compiler in implementation-defined ways.

```

base_domain_declaration ::=
    [extended] base domain identifier_1
    [(base_domain_parameter_list)]
is
    patterns
    options
end [identifier_2];

base_domain_parameter_list ::=
    base_domain_parameter {; base_domain_parameter}

```

1. If present, identifier_2 shall be equivalent to identifier_1. Identifier_1 is the name of the base domain.
2. The keyword **extended** may appear in a base_domain_declaration only if it also appears in the enclosing module declaration.

7.1.1.1 Base Domain Parameters

```

base_domain_parameter ::=
    identifier : data_class [ := static_expression] |
    map := pos |
    map := image

```

```

data_class ::=
    integer | character |
    fixed | float |
    enumeration

```

1. The Ada identifiers within the list of `base_domain_parameters` of a `base_domain_declaration` are the names of the parameters that may appear in a `parameter_association_list` within a `domain_declaration` based on this base domain (see 7.1.3 of this specification). The `static_expression` within a `base_domain_parameter`, when present, specifies a default value for the parameter. This default value shall be of the correct `data_class`; that is, in the parameter declaration

```
Id : dcl := expr;
```

where *dcl* is a `data_class`, `DATACLASS(expr)` shall be *dcl*. Further, `DATACLASS(Id)` is *dcl*, whether or not the initializing expression *expr* is present, and `DOMAIN(Id)` is `NO_DOMAIN`.

2. A base domain is classified by its `data_class`. That is, an enumeration base domain is a base domain whose `data_class` is **enumeration**, a fixed base domain is a base domain whose `data_class` is **fixed**, etc.
3. Every enumeration base domain has two predefined parameters: **enumeration** and **map**. These parameters are special in that the values that are assigned to them by a domain declaration (see 7.1.3 of this specification) are not of any of the data classes listed above. The value of an **enumeration** parameter is an `enumeration_reference` (see 6.4 of this specification); the value of **map** is a `database_mapping` (see 7.1.3 of this specification). A base domain declaration may explicitly declare a **map** parameter for the purpose of assigning a default mapping. An enumeration base domain shall not redefine the predefined `base_domain_parameter` **enumeration**.

There are two possible default mappings: **pos** and **image**. The value **pos** specifies that the Ada predefined attribute function 'POS of the Ada type corresponding to the `enumeration_reference`, which is the **enumeration** parameter value in the domain declaration, shall be used to translate enumeration literals to their database encodings. Similarly for **image** and the 'IMAGE attribute. See annex A of ISO 8652:1987 and 7.1.3 and 7.3 of this specification.

Every enumeration base domain whose **dbms type** is **char** or **character** shall have a third predefined parameter, **length** whose value is an integer of an implementation defined range. Such an enumeration base domain shall not redefine the predefined `base_domain_parameter` **length**.

4. Every fixed base domain has a predefined parameter **scale** whose value is an integer of an implementation defined range (see 6.1 of ISO/IEC 9075:1992). A fixed base domain shall not redefine the predefined `base_domain_parameter` **scale**.
5. Every character base domain has a predefined parameter **length** whose value is an integer of an implementation defined range (6.1 of ISO/IEC 9075:1992). A character base domain shall not redefine the predefined `base_domain_parameter` **length**.

7.1.1.2 Base Domain Patterns

The patterns portion of a base domain declaration forms a template for the generation of Ada text, which forms the Ada semantics of domains based on the given base domain.

```

patterns ::=
    {pattern}

pattern ::=
    domain_pattern | subdomain_pattern | derived_domain_pattern

domain_pattern ::=

```

```

domain pattern is pattern_list
end pattern;

subdomain_pattern ::=
  subdomain pattern is pattern_list
end pattern;

derived_domain_pattern ::=
  derived domain pattern is pattern_list
end pattern;

pattern_list ::=
  pattern_element {pattern_element}

pattern_element ::=
  character_literal

```

Patterns are used to create the Ada constructs that implement the Ada semantics of a domain, subdomain, or derived domain declaration (see 7.1.3 of this specification). Patterns are considered templates; parameters within a pattern are replaced by the values assigned to them either in the domain declaration, by inheritance, or by default. See 7.1.3 of this specification.

For a parameter to be recognized as such in a pattern, it is enclosed in square brackets ([,]). For the purpose of pattern substitution, a base domain may use a parameter **self**. When a pattern is instantiated, **self** is the result of applying the transformation AdaID to the name of the domain or subdomain being declared (see 5.3 of this specification). A base domain may use a parameter **parent** for the purpose of pattern substitution in a subdomain_pattern or a derived_domain_pattern. When such a pattern is instantiated, **parent** is AdaID applied to the name of the parent domain (see 5.3 and 7.1.3 of this specification).

Within a given character_literal of a pattern, a substring contained in matching curly brackets ({,}) is an optional phrase. Optional phrases may be nested. An optional phrase appears in the instantiated template if all parameters within the phrase have values assigned by a domain declaration (see 7.1.3 of this specification); the phrase does not appear when none of the parameters within the phrase has an assigned value. If some but not all parameters within an optional phrase have values assigned by a given domain declaration, the declaration is in error.

7.1.1.3 Base Domain Options

```

options ::=
  {option}

option ::=
  fundamental |
  for word_list use pattern_list ;

fundamental ::=
  for not null type name use pattern_list; |
  for null type name use pattern_list; |
  for data class use data_class; |
  for dbms type use dbms_type [pattern_list]; |
  for conversion from type to type use converter ;

dbms_type ::= int | integer | smallint |
  real | double precision |
  char | character |

```

```

implementation defined

type ::= dbms | not null | null

converter ::=
    function pattern_list |
    type mark

word_list ::=
    implementation defined

```

Options are used to define aspects of base domains that are essential to the declaration of domains within the SAMeDL. The fundamental options are required. Implementations may add options beyond those given above. The meanings of the fundamental options are given by the following list.

1. The **null** and **not null** type names are the targets of the function AdaTYPE. They are the names of the types of parameters and parameter components in Ada procedures. See 8.6 and 8.7 of this specification.
2. The **data class** option specifies the data class (see 7.1.1.1 of this specification) of all objects of any domain based on this base domain. If *BD* is a base domain to which the data class *dc* is assigned by an option in its definition, and if *D* is a domain based directly or indirectly (see 7.1.3 of this specification) on *BD*, then DATACLASS(*D*)=*dc*. The data class governs the use of literals with such objects (see 8.8 and 8.10 of this specification).
3. The **dbms type** of a base domain is the *SQL_data_type* (see 6.1 of ISO/IEC 9075:1992) to be used when declaring parameters of the concrete interface (SQL module) for all objects of domains based directly or indirectly on the base domain. See 8.6, 8.7, and 8.8 of this specification. If the dbms type of a base domain is implementation defined, the keyword **extended** shall appear in the declaration of the base domain.
4. An operand of the **conversion** option is a means of converting non-null data between objects of the not null-bearing type, the null-bearing type (see 7.1.3 of this specification) and the dbms type associated with a domain. A method shall be a function, an attribute of a type, or a type conversion. A means of determining the identity of these methods shall appear in the options of a base domain. The identity of a method may be given as a pattern containing parameters.

However, enumeration domains do not have converters between the dbms type and the not null-bearing type, as the **map** parameter predefined for all enumeration domains describes a conversion method between enumeration and database representations of non-null data. The method is the application, as appropriate, of the function described by the database_mapping that is the operand of the map parameter association. See 7.1.1.1 and 7.1.3 of this specification.

7.1.2 The SAME Standard Base Domains

The predefined definitional module, SAMeDL_Standard, contains the declarations of the predefined SAME Standard Base Domains: SQL_Int, SQL_Smallint, SQL_Char, SQL_Real, SQL_Double_Precision, SQL_Enumeration_as_Char, and SQL_Enumeration_as_Int. The text of SAMeDL_Standard appears in Annex A of this specification.

7.1.3 Domain and Subdomain Declarations

```

domain_declaration ::=
    domain identifier is new bas_dom_ref [not null]
        [ ( parameter_association_list ) ] ;

subdomain_declaration ::=
    subdomain identifier is dom_ref [not null]
        [ ( parameter_association_list ) ] ;

dom_ref ::=
    domain_reference | subdomain_reference

bas_dom_ref ::=
    dom_ref | base_domain_reference

parameter_association_list ::=
    parameter_association {, parameter_association}

parameter_association ::=
    identifier => static_expression |
    map => database_mapping |
    enumeration => enumeration_reference |
    scale => static_expression |
    length => static_expression

database_mapping ::=
    enumeration_association_list | pos | image

enumeration_association_list ::=
    ( enumeration_association {, enumeration_association} )

enumeration_association ::=
    enumeration_literal => database_literal

```

1. Consider the domain declaration:

domain DD is new EE ...

- a. If EE is a base_domain_reference, then EE is said to be the base domain of DD.
- b. Otherwise, EE is a domain_reference or subdomain_reference, the base domain of DD is defined to be the base domain of EE, DD is said to be derived from EE, and EE is said to be the parent of DD.

2. Similarly, in the subdomain declaration

subdomain FF is GG ...

the base domain of FF is defined as the base domain of GG, FF is said to be a subdomain of GG, and GG is said to be the parent of FF.

3. The database type of a domain *D*, denoted as DBMS_TYPE(*D*), is the value, appropriately parameterized, of the **for dbms type** option from the base domain of *D*. See 7.1.1.3 of this specification.
4. The data class of a domain *D*, denoted as DATACLASS(*D*), is the data class of its base domain, the value of the **for data class** option. A domain is numeric if its data class is numeric (see 7.1.1.1 of this specification).
5. Except for **scale**, **enumeration**, **length**, and **map**, an identifier within a parameter_association shall be the name of a base_domain_parameter in the declaration of the base domain of the domain or subdomain being declared. See 7.1.1.1 of this specification.

6. A domain or subdomain D is said to *assign the expression E* to the parameter P , if
- a. the parameter_association $P \Rightarrow E$ appears in the declaration of D ; or
 - b. (a) does not hold, D is a subdomain or a derived domain, and the parent domain assigns the expression E to the parameter P ; or
 - c. (a) and (b) do not hold and in the base_domain_declaration for the base domain of D , the base_domain_parameter

$$P : class := E$$
 appears.

In all cases, $DATACLASS(E)$ shall be $DATACLASS(P)$ as defined by the declaration of the base domain. See 7.1.1.1 of this specification.

7. If a domain D assigns the expression E to a parameter P , then

- $DOMAIN(D.P) = NO_DOMAIN$.
- $DATACLASS(D.P) = DATACLASS(E)$
- $LENGTH(D.P) = LENGTH(E)$
- $SCALE(D.P) = SCALE(E)$

8. A domain_declaration shall assign an expression to each base_domain_parameter that appears in any non-optional phrase

- of the base domain's domain_pattern, if the declaration is not declaring a derived domain;
- of the base domain's derived_domain_pattern, if the declaration is the declaration of a derived domain.

Similar rules govern subdomain_declarations and subdomain_patterns. See 7.1.1.2 of this specification.

9. The scale of a domain D , denoted $SCALE(D)$, is defined by

- if D is not a numeric domain, $SCALE(D) = NO_SCALE$;
- if D is an integer domain, $SCALE(D) = 0$;
- if D is a float domain, $SCALE(D)$ = a value greater than the scale of any non-float domain or object;
- if D is a fixed domain, $SCALE(D)$ = the value assigned by D to the **scale** base_domain_parameter.

The value assigned to the **scale** parameter in the declaration of a fixed domain shall be an integer from an implementation defined range.

10. The length of a domain D , denoted $LENGTH(D)$, is defined by

- if D is not a character domain, $LENGTH(D) = NO_LENGTH$;
- if D is a character domain, $LENGTH(D)$ = the value assigned by D to the **length** base_domain_parameter.

The value assigned to the **length** parameter in the declaration of a character domain shall be an integer from an implementation defined range.

11. Any `domain_declaration` or `subdomain_declaration` of an enumeration domain shall assign an `enumeration_reference` as to the `base_domain_parameter` **enumeration** and a `database_mapping` to the `base_domain_parameter` **map**. If the **map** parameter is assigned an `enumeration_association_list`, then

a. Each `enumeration_literal` within the enumeration referenced by the `enumeration_reference` given by the **enumeration** parameter shall appear as the `enumeration_literal` of exactly one `enumeration_association`.

b. No `database_literal` shall appear in more than one `enumeration_association`.

Note: These constraints ensure that the `database_mapping` is an invertible (i.e., one-to-one) function. That function is used for both compile time and runtime data conversions. See 7.1.1.3 and 7.3 of this specification.

12. Let D be an enumeration domain or subdomain declaration and let En be the name of the enumeration referenced by the value assigned by D to the **enumeration** `base_domain_parameter`. D is said to *assign the expression* E to the enumeration literal El , if D assigns the `database_mapping` M as the value of the **map** `base_domain_parameter` and M

-- is **pos**, and $E = En'Pos(El)$, or

-- is **image**, and $E = En'Image(El)$, or

-- is an `enumeration_association_list` containing an `enumeration_association` of the form $El \Rightarrow E$

See 7.1.6 of this specification.

13. The `database_mapping` of an enumeration domain or subdomain declaration D should preserve the ordering implied by that domain's `enumeration_reference` ER . That is, if L_1 and L_2 are enumeration literals of ER such that L_1 occurs before L_2 in ER 's `enumeration_literal_list`, then the value assigned to L_1 by D should be less than the value assigned to L_2 by D .

14. A domain or subdomain is said to be *not null only* if it or any of its parent domains is declared with the **not null** phrase. In that case no object of the domain can contain the null value.

Ada Semantics

An instantiation of a pattern defined for the base domain of the domain being declared, as described in 7.1.1.2 of this specification, shall appear within the Ada package specification corresponding to the module within which the `domain_declaration` appears. If in the `domain_declaration`:

`domain DD is new EE ...`

EE is a `base_domain_reference`, then the `domain_pattern` is instantiated; if EE is a `domain_reference`, the `derived_domain_pattern` is instantiated; for the `subdomain_declaration`

`subdomain FF is GG ...`

the `subdomain_pattern` is used.

Note: Examples

The following examples illustrate the declaration of domains and have been annotated with references to the appropriate clauses of the language definition. The base domains used in these examples exist in the predefined definitional module `SAMeDL_Standard`, which appears in Annex A of this specification. The constant `Max_SQL_Int` is declared in the predefined definitional module `SAMeDL_System`. Both `SAMeDL_Standard` and `SAMeDL_System` (see Annex B of this specification) are assumed to be visible, as is the enumeration declaration `Colors` (see 7.1.6 of this specification).

26 Database Programming Language - SAMeDL

```

domain Weight is new SQL_Int ( -- 7.1.3: #1a
    First => 0,                -- 7.1.3: #5 and #8
    Last  => Max_SQL_Int);     -- 7.1.3: #5 and #8

domain Weight_In_Pounds is new Weight;
domain Weight_In_Grams  is new Weight;

domain City_Names is new SQL_Char ( -- 7.1.3: #1a
    Length => 15);             -- 7.1.3: #5 and #10

domain Colors is new SQL_Enumeration_As_Char ( -- 7.1.3: #1a
    enumeration => Color_Values,                -- 7.1.3: #11
    map          => image);                     -- 7.1.3: #11 and #12

domain Auto_Weight is new Weight ( -- 7.1.3: #1b
    Last => 10000);                -- 7.1.3: #5

subdomain Auto_Part_Weight is Auto_Weight ( -- 7.1.3: #2
    Last => 2000);                 -- 7.1.3: #5 and #8

```

These declarations produce the following Ada code.

```

-- the Ada code below is the instantiation of the domain pattern
-- from base domain SQL_Int

type Weight_Not_Null is new SQL_Int_Not_Null
    range 0 .. implementation_defined;
type Weight_Type is new SQL_Int;
package Weight_Ops is new SQL_Int_Ops (
    Weight_Type, Weight_Not_Null);

-- The domains Weight_In_Pounds and Weight_In_Grams are not illustrated
-- here as there instantiations are essentially identical to Weight.

-- the Ada code below is the instantiation of the domain pattern
-- from base domain SQL_Char

type City_NamesNN_Base is new SQL_Char_Not_Null;
subtype City_Names_Not_Null is City_NamesNN_Base (1 .. 15);
type City_Names_Base is new SQL_Char;
subtype City_Names_Type is City_Names_Base (City_Names_Not_Null'Length);
package City_Names_Ops is new SQL_Char_Ops(
    City_Names_Base, City_NamesNN_Base);

-- the Ada code below is the instantiation of the domain pattern
-- from the base domain SQL_Enumeration_As_Char

package Colors_Pkg is new SQL_Enumeration_Pkg(Color_Values);
type Colors_Type is new Colors_Pkg.SQL_Enumeration;

-- the Ada code below is the instantiation of the derived domain
-- pattern from the base domain SQL_Int

type Auto_Weight_Not_Null is new Weight_Not_Null
    range Weight_Not_Null'First .. 10000;
type Auto_Weight_Type is new Weight_Type;
package Auto_Weight_Ops is new SQL_Int_Ops(
    Auto_Weight_Type, Auto_Weight_Not_Null);

-- the Ada code below is the instantiation of the subdomain
-- pattern from the base domain SQL_Int

subtype Auto_Part_Weight_Not_Null is Auto_Weight_Not_Null
    range Auto_Weight_Not_Null'First .. 2000;
type Auto_Part_Weight_Type is new Auto_Weight_Type;
package Auto_Part_Weight_Ops is new SQL_Int_Ops(
    Auto_Part_Weight_Type, Auto_Part_Weight_Not_Null);

```

End Examples

7.1.4 Constant Declarations

```
constant_declaration ::=
    constant identifier [: domain_reference]
    is static_expression ;

static_expression ::=
    value_expression
```

A static expression is a value expression (see 8.10 of this specification) whose value can be calculated at compile time; i.e., whose leaves are all either literals or constants.

Now let K denote the constant declaration

```
constant C [: D ] is E ;
```

1. DATACLASS(K) is DATACLASS(E), the data class of the expression E . See 5.4 and 8.10 of this specification.
2. If DATACLASS(K) is **enumeration**, then D shall be present and shall name an enumeration domain of which E is an enumeration literal.
3. If DATACLASS(K) is **character**, then D shall be present.
4. If the domain_reference D is not present, then
 - a. C is a *universal* constant of class DATACLASS(K).
 - b. AdaTYPE(K) is an anonymous type, *universal_T*, where T is DATACLASS(K).
 - c. If DATACLASS(K) is numeric, then SCALE(K) = SCALE(E).
 - d. DOMAIN(K) = NO_DOMAIN
5. If the domain_reference D is present, then
 - a. DOMAIN(K)= D and E shall conform to D (see 6.5 of this specification).
 - b. If DATACLASS(K) is numeric, then SCALE(K) = SCALE(D), and SCALE(E) shall not exceed SCALE(D).
 - c. If DATACLASS(K) is character, then LENGTH(K) = LENGTH(D) and LENGTH(E) shall not exceed LENGTH(D).
 - d. AdaTYPE(K) is defined as the type name within D designated as not null-bearing. That type name shall not refer to a limited type.

Ada Semantics

Let VALUE represent the function which calculates the value of a static_expression. Let SE be a static_expression. VALUE(SE) is given recursively as follows:

1. If SE contains no operators, then
 - a. If SE is a database_literal, then VALUE(SE) = SE .

28 Database Programming Language - SAMeDL

b. If SE is an enumeration_literal of domain D , and D assigns expression E to that enumeration literal, then $VALUE(SE) = E$.

c. If SE is a reference to the constant whose declaration is given by.

constant C [$: D$] **is** E ;

then $VALUE(SE) = VALUE(E)$.

d. If SE is a reference to a parameter P from domain D , and D assigns the expression E to P , then $VALUE(SE) = VALUE(E)$.

2. If SE is $D(SE_1)$, where D is a domain name, then $VALUE(D(SE_1)) = VALUE(SE_1)$.

3. If SE is $+SE_1$ (or $-SE_1$) then $VALUE(SE) = +VALUE(SE_1)$ (or $-VALUE(SE_1)$).

4. If SE is SE_1 *op* SE_2 , where *op* is an arithmetic operator, then $VALUE(SE) = VALUE(SE_1)$ *op* $VALUE(SE_2)$ where *op* is evaluated according to the rules of SQL. See 6.12 of ISO/IEC 9075:1992.

5. If SE is (SE_1) then $VALUE(SE) = (VALUE(SE_1))$.

Again, let K denote the constant declaration

constant C [$: D$] **is** E ;

If E is not an enumeration literal, let Q be the Ada representation of $VALUE(E)$; otherwise, let Q be $AdaID(E)$. Then the Ada library unit package specification corresponding to the module in which the constant declaration K appears shall have an Ada constant declaration of the form

$AdaID(C)$: **constant** [$AdaTYPE(K)$] := Q ;

The type designator $AdaTYPE(K)$ is omitted from this declaration if it is an anonymous type.

Note: Examples

The following are examples of constant declarations.

```
constant Grams_In_Pound is 453.59237;
constant The_Color_Red : Colors is Red;
constant Home_Port : City_Names is 'Pittsburgh';
```

Note that Grams_in_Pound is a universal constant of class **fixed**. These declarations generate the following Ada declarations.

```
Grams_In_Pound : constant := 453.59237;
The_Color_Red : constant : Color_Values := Red;
Home_Port : constant City_Name_Not_Null := "Pittsburgh" ;
```

End Examples

7.1.5 Record Declarations

```
record_declaration ::=
    record identifier_1 [named_phrase] is
        component_declarations
    end [identifier_2] ;

named_phrase ::=
    named identifier

component_declarations ::=
    component_declaration {component_declaration}

component_declaration ::=
    component {, component} : domain_reference [not null] ;
```

```

component ::=
    component_name [dblength [named_phrase]]

component_name ::= Ada_identifier

```

If present, identifier_2 shall be equivalent to identifier_1. Identifier_1 is the *name* of the record.

Let R be a record declaration. Define $\text{AdaNAME}(R)$ to be

1. The alias N , if the named_phrase **named** N appears in the declaration.
2. *Row*, otherwise.

Note: $\text{AdaNAME}(R)$ is the default for the name of the row record formal parameter in the parameter profile of any procedure that uses the declaration R . See 8.2, 8.5, and 8.9 of this specification.

Ada Semantics

The Ada library unit package specification corresponding to the module within which the record_declaration R appears shall have an Ada record type declaration R_{Ada} defined as follows:

1. The name of the record type R_{Ada} shall be $\text{AdaID}(\text{identifier_1})$.
2. For some integer k , let the component_declarations of R be given by the sequence

$\text{components}_{i_j} : D_{i_j} [\text{not null}_{i_j}]$

for $1 \leq i \leq k$, where components_i is given by the sequence

$C_{i_j} [\text{dblength}_{i_j} [\text{named } N_{i_j}]]$

where $1 \leq j \leq m_i$ for some integer m_i . R_{Ada} shall be equivalent, in the sense of 3.2.10 and 3.7.2 of ISO 8652:1987, to a record type whose components are given by the sequence

$\text{COMP}_{i_j} [\text{DBleng}_{i_j}]$

where i and j are bound as before and COMP_{i_j} is given by

$\text{AdaID}(C_{i_j}) : T_{i_j} ;$

where T_{i_j} is an Ada type name determined to be:

- a. The not null-bearing type name within the domain D_{i_j} , if either D_{i_j} is a not null only domain or **not null** _{i_j} is present in R ;
- b. Otherwise the null-bearing type name within the domain D_{i_j} .

The optional component DBleng_{i_j} appears if the optional **dblength** _{i_j} phrase appears and in that case is of the form

$\text{DBLngNAME}_{i_j} : \text{Ada_Indicator_Type} ;$

where DBLngNAME_{i_j} is $\text{AdaID}(N_{i_j})$ if N_{i_j} appears and is $\text{AdaID}(C_{i_j})_DbLength$, otherwise; and $\text{Ada_Indicator_Type}$ is the type $\text{SQL_Standard.Indicator_Type}$ (see 12.3.8.a.iii of ISO/IEC 9075:1992).

Note: Examples

The following example illustrates the declaration of a record object.

```

record Parts_Row_Record_Type named Parts_Row_Record is
    Part_Number      : Part_Numbers not null;
    Part_Name        : Part_Names;
    Color            : Colors;
    Weight           : Weight_In_Grams;
    City             : City_Names;

```

```
end Parts_Row_Record_Type;
```

This declaration produces the following Ada code. It has been annotated with references to the appropriate clauses of the language definition.

```
type Parts_Row_Record_Type is record -- Ada Semantics #1
  Part_Number      : Part_Numbers_Not_Null; -- Ada Semantics #2
  Part_Name        : Part_Names_Type;
  Color            : Colors_Type;
  Weight           : Weight_In_Grams_Type;
  City             : City_Names_Type;
end record;
```

End Examples

7.1.6 Enumeration Declarations

Enumerations are used to declare sets of enumeration literals for use in enumeration domains and status maps. See 7.1.3 and 7.1.8 of this specification.

```
enumeration_declaration ::=
  enumeration identifier_1 is ( enumeration_literal_list ) ;

enumeration_literal_list ::=
  enumeration_literal {, enumeration_literal}
```

1. Identifier_1 is the *name* of the enumeration.
2. Each identifier within an enumeration_literal_list is said to be an enumeration literal of the enumeration. The enumeration_declaration is considered to declare each of its enumeration_literals. An enumeration_literal may appear in multiple enumeration_declarations.

Ada Semantics

There shall be, within the Ada package specification corresponding to the module within which an enumeration appears, a type declaration of the form

```
type AdaID(identifier_1) is ( AdaID(enumeration_literal_list) ) ;
```

The application of the transformer AdaID (see 5.3 of this specification) to a list is accomplished by application of the transformer to each element of the list. Ada character literals shall not be used in enumerations. See 5.4 of this specification.

Note: Examples

The following are examples of enumeration declarations.

```
enumeration Single_Row_Status is (
  More_Than_One_Row, No_Such_Row, Row_Found);

enumeration Color_Values is (
  Purple, Blue, Green, Yellow, Orange, Red, Black, White);
```

The above declarations produce the following Ada code.

```
type Single_Row_Status is (
  More_Than_One_Row, No_Such_Row, Row_Found);

type Color_Values is (
  Purple, Blue, Green, Yellow, Orange, Red, Black, White);
```

End Examples

7.1.7 Exception Declarations

```
exception_declaration ::=
    exception identifier ;
```

Identifier is the *name* of the exception.

Ada Semantics

There shall be, within the Ada package specification corresponding to the module within which an exception declaration appears, an exception declaration of the form

```
AdaID(identifier_1) : exception ;
```

Note: Examples

The following are examples of exception declarations.

```
exception Data_Definition_Does_Not_Exist;

exception Insufficient_Privilege;

exception Transaction_Rollback;
```

The above declarations produce the following Ada code.

```
Data_Definition_Does_Not_Exist : exception;

Insufficient_Privilege : exception;

Transaction_Rollback : exception;
```

End Examples

7.1.8 Status Map Declarations

The execution of any procedure (see 6.7, 8.2, and 8.5 of this specification) causes the execution of an SQL procedure. That execution causes a special parameter, called the SQL status parameter, to be “set to a status code that either indicates that a call of the procedure completed successfully or that an exception condition occurred during execution of the procedure” (See 4.18.1 of ISO/IEC 9075:1992). Status maps are used within abstract modules to process the status data in a uniform way. Each map declares a partial function from the set of all possible SQL status parameter values onto (1) enumeration literals of an enumeration and (2) raise statements.

```
status_map_declaration ::=
    [ sqlcode | sqlstate ] status identifier_1
    [named_phrase]
    [uses target_enumeration]
    is ( status_assignment { , status_assignment } );

target_enumeration ::=
    enumeration_reference | boolean

status_assignment ::=
    left_hand_side => enumeration_literal |
    left_hand_side => raise exception_reference

left_hand_side ::=
    static_expr { , static_expr }

static_expr ::=
    static_expression |
    static_expression .. static_expression
```

(see 3.5 of ISO 8652:1987)

1. Identifier_1 is the *name* of the map.
2. A target_enumeration of **boolean** is a reference to the predefined Ada enumeration type Standard.boolean.
3. If the optional **uses** clause is not present, then only status_assignments that contain **raise** shall be present in the status_map_declaration.
4. Every *Ada_enumeration_literal* within a status_assignment shall be an *Ada_enumeration_literal* within the enumeration referenced by the target_enumeration.
5. If neither **sqlcode** nor **sqlstate** is specified, then **sqlstate** is assumed.
6. If **sqlcode** is specified, then if E is a static_expression appearing in a left_hand_side, then DATACLASS(E) = **integer**. In this case a static_expr of the form $E_1 \dots E_2$ shall be permitted in a left_hand_side, provided $E_1 \leq E_2$. If $v_1, v_2, \dots, v_m$ are all the integers between E_1 and E_2 inclusive, then the form $E_1 \dots E_2$ is equivalent to the list of static expressions $v_1, v_2, \dots, v_m$.
7. If **sqlstate** is specified, then if E is a static_expression appearing in a left_hand_side, then DATACLASS(E) = **character**, and either
 - a. $\text{LENGTH}(E) = 5$, and, if $\text{DOMAIN}(E) \neq \text{NO_DOMAIN}$, then $\text{DOMAIN}(E) = \text{SAMEDL_Standard.SQLSTATE_Domain}$; or
 - b. $\text{LENGTH}(E) = 2$, and, if $\text{DOMAIN}(E) \neq \text{NO_DOMAIN}$, then $\text{DOMAIN}(E) = \text{SAMEDL_Standard.SQLSTATE_Class_Domain}$. If $v_1, v_2, \dots, v_m$ are all the SQLSTATE values whose Class value is given by the value of E, then E is equivalent to the list of static expressions $v_1, v_2, \dots, v_m$. See annex A for the text of the predefined module SAMeDL_Standard. See 22.1 of ISO/IEC 9075:1992 for the standard values of SQLSTATE. Other values of SQLSTATE may be defined by an implementation.
8. If E and E' are distinct static expressions appearing in the clauses of a status map (possibly via the equivalence of previous rules), then E and E' shall not evaluate to the same value.

Notes: **sqlstate** is the preferred form. **sqlcode** is a feature deprecated in ISO/IEC 9075:1992, annex D.

A status_assignment takes the form of a list of alternatives as found in Ada case statements, aggregates, and representation clauses.

The **others** choice is not valid for status_assignments, however.

SAMeDL_Standard contains the definition of a status map Standard_Map, defined as follows:

```

sqlstate status Standard_Map
  named Is_Found
  uses boolean
is
  (Successful_Completion_No_Subclass => True,
   No_Data_No_Subclass => False);

```

Standard_Map is the status map for those fetch statements that appear in cursor declarations by default. (See 8.5 of this specification.) It signals end of table by returning false.

Note: Examples

The following is an example of a status map declaration. For the enumeration and exception declarations, refer to the examples in 7.1.6 and 7.1.7 of this specification. They are assumed to be visible at the point at which the status map is declared. Further, direct visibility (i.e., **use**) of SAMeDL_Standard is assumed.