# Database Programming Language --
# The SQL Ada Module Description Language
# SAMeDL

# 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.

Draft International Standards adopted by the technical committees are circulated to the member bodies for approval before their acceptance as International Standards by the ISO Council. They are approved in accordance with ISO procedures requiring at least 75% approval by the member bodies voting.

International Standard ISO/IEC 12227:1994, *Database Programming Language -- The SQL Ada Module Description Language SAMeDL*, was prepared by Joint Technical Committee ISO/IEC JTC1, *Information Processing Systems.*

This International Standard contains six annexes of which three are normative and three informative:

-- Annex A (normative) : SAMeDL_Standard

-- Annex B (normative) : SAMeDL_System

-- Annex C (normative) : Standard Base Domain Operations and Specifications

-- Annex D (informative) : Transform Chart

-- Annex E (informative) : Glossary

-- Annex F (informative) : Bibliography

# Introduction

The organization of this International Standard is as follows:

1. Clause 1, ‘‘Scope,’’ specifies the scope of this International Standard.

2. Clause 2, ‘‘Normative References,’’ identifies additional International Standards that, through reference in this International Standard constitute provisions of this International Standard.

3. Clause 3, ‘‘Notations and Structures,’’ describes the syntax notation used in and the structure of the clauses of this International Standard.

4. Clause 4, ‘‘Design Goals and Language Summary,’’ presents an overview of the SQL Ada Module Description Language.

5. Clause 5, ‘‘Lexical Elements,’’ defines the lexical elements of the language.

6. Clause 6, ‘‘Common Elements,’’ defines concepts and syntactic categories which are used in the definition of further concepts and syntactic categories of the language.

7. Clause 7, ‘‘Data Description Language and Data Semantics,’’ defines that portion of the SQL Ada Module Description Language in which database description and other declarative items are encoded.

8. Clause 8, ‘‘Abstract Module Description Language,’’ defines that portion of the SQL Ada Module Description Language in which database manipulation procedures are encoded.

9. Clause 9, ‘‘Conformance,’’ defines conformance criteria.

10. Annex A, ‘‘SAMeDL_Standard,’’ is a normative Annex containing the text of the predefined SAMeDL Definition Module SAMeDL_Standard.

11. Annex B, ‘‘SAMeDL_System,’’ is a normative Annex containing the text of the predefined SAMeDL Definition Module SAMeDL_System. Portions of this module are implementation defined.

12. Annex C, ‘‘Standard Support Package Specifications,’’ is a normative Annex containing Ada text declaring the standard support packages.

13. Annex D, ‘‘Transform Chart,’’ is an informative annex giving in a tabular form a description of string-to-string transformation functions used in the definition of the SAMeDL.

14. Annex E, ‘‘Glossary,’’ is an informative annex giving explanations of terms used in this International Standard.

15. Annex F is a Bibliography.

This International Standard is based on and defined with respect to the International Standards for *Database Language SQL*, ISO 9075:1992 and *Programming Language Ada*, ISO 8652:1987 (endorsement by ISO of ANSI/MIL-STD-1815A-1983 and AFNOR NF Z 65-700).

# 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, *Database Language SQL*.

-- ISO 8652:1987, *Programming Language -- Ada* (endorsement by ISO of ANSI-MIL-STD-1815A-1983 and AFNOR NF Z 65-700).

# 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.

### 4.2.3 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.

### 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, than 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 LENGTH(*CL*) in the sense of 5.3, Syntax Rule 5 in ISO/IEC 9075:1992. For any non character literal, *L*, LENGTH(*L*)=NO_LENGTH.

4. Integer, fixed and float literals are collectively known as numeric literals. Every numeric literal *NL* has a scale, 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 SCALE(*L*)=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** $N_1$];
   **with schema** S [**as** $N_2$];

   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 $N_1$ is the exposed name of M. Similar comments apply to S and $N_2$. 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 (**with**ed 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 *Id*. Then *Id* is a reference to the item *Im* if *Im* is not a module, procedure, cursor or cursor procedure, or table and

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

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

*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 Curse1
      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*) $\neq$ 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
```

```
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 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* => *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 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* => *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).

```
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*.

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 unversal 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 AdaNAME(*R*) to be

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

2. *Row*, otherwise.

   *Note*: 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 AdaID(identifier_1).

2. For some integer *k*, let the component_declarations of *R* be given by the sequence

   $\texttt{components}_{\texttt{i}}$ : $\texttt{D}_{\texttt{i}}$ [**not null**$_{\texttt{i}}$]

   for $1 \leq i \leq k$, where $\text{components}_i$ is given by the sequence

   $\texttt{C}_{\texttt{i}_\texttt{j}}$ [**dblength**$_{\texttt{i}_\texttt{j}}$ [**named** $\texttt{N}_{\texttt{i}_\texttt{j}}$]]

   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

   $\texttt{COMP}_{\texttt{i}_\texttt{j}}$ [$\texttt{DBleng}_{\texttt{i}_\texttt{j}}$]

   where *i* and *j* are bound as before and $\text{COMP}_{i_j}$ is given by

   $\texttt{AdaID(C}_{\texttt{i}_\texttt{j}}\texttt{)}$ : $\texttt{T}_{\texttt{i}}$ ;

   where $T_i$ is an Ada type name determined to be:

   a. The not null-bearing type name within the domain $D_i$, if either $D_i$ is a not null only domain or **not null**$_i$ is present in *R*;

   b. Otherwise the null-bearing type name within the domain $D_i$.

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

   $\texttt{DBLngNAME}_{\texttt{i}_\texttt{j}}$ : Ada_Indicator_Type ;

   where $\text{DBLngNAME}_{i_j}$ is $\text{AdaID}(N_{i_j})$ if $N_{i_j}$ appears and is $\text{AdaID}(C_{i_j})\text{\_DbLength}$, otherwise; and Ada_Indicator_Type is the type 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_Rollbasck : 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 $E_1$ .. $E_2$ shall be permitted in a left_hand_side, provided $E_1 \le E_2$. If $v_1$, $v_2$ ,..., $v_m$ are all the integers between $E_1$ and $E_2$ inclusive, then the form $E_1$ .. $E_2$ is equivalent to the list of static expressions $v_1$, $v_2$ ,..., $v_m$.

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. LENGTH(E) = 5, and, if DOMAIN(E) ≠ NO_DOMAIN, then DOMAIN(E) = SAMeDL_Standard.SQLSTATE_Domain; *or*

   b. LENGTH(E) = 2, and, if DOMAIN(E) ≠ NO_DOMAIN, then DOMAIN(E) = SAMeDL_Standard.SQLSTATE_Class_Domain. If $v_1$, $v_2$ ,..., $v_m$ 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 $v_1$, $v_2$ ,..., $v_m$. 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.

This map might be usefully applied to single row select statements. It distinguishes the excepitonal condition of retrieving more than one row from the condition of retrieving no rows. It eliminates the distinction between a successful completion and a completion in which a warning was returned. It raises an exception in the Ada application in two subcases of an SQL transaction rollback. See 6.6 and Annex A of this specification and 22 in ISO/IEC 9075:1992.)

```
status Single_Row_Select_Map
  named Result
  uses Single_Row_Status is (
    Cardinality_Violation_No_Subclass => More_Than_One_Row,
    No_Data_No_Subclass               => No_Such_Row,
    Succesful_Completion_No_Subclass,
      Warning                         => Row_Found,
    Transaction_Rollback_No_Subclass,
     Transaction_Rollback_Serialization_Failure => raise Transaction_Rollback);
```

**End Examples**

## 7.2 Schema Modules

```
schema_module ::=
    [context]
    [extended]schema module identifier_1 is
        {schema_element}
    end [identifier_2];

schema_element ::=
    table_definition | view_definition | SQL_grant_statement |
    extended_schema_element

SQL_grant_statement ::=
     (see 11.36 of ISO/IEC 9075:1992)
```

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

2. Identifier_1 shall be different from any other schema module name (See clause 11.1 of ISO/IEC 9075:1992).

3. An extended_schema_element may appear in a schema_module only if the keyword **extended** appears in the associated schema module declaration.

### 7.2.1 Table Definitions

```
table_definition ::=
    [extended]table identifier_1 is
        table_element {, table_element}
    end [identifier_2] ;

table_element ::=
    column_definition | table_constraint_definition |
    extended_table_element

column_definition ::=
    SQL_column_name [SQL_data_type]
        [SQL_default_clause]
         [column_constraint] : domain_reference

SQL_default_clause ::=
    (see 11.5 in ISO/IEC 9075:1992)

column_constraint ::=
```

```
    not null [SQL_unique_specification] |
    SQL_reference_specification |
    check ( search_condition )

SQL_unique_specification ::=
    (see 11.7 in ISO/IEC 9075:1992)

SQL_reference_specification ::=
    (see 11.8 in ISO/IEC 9075:1992)

table_constraint_definition ::=
    SQL_unique_constraint_definition |
    SQL_referential_constraint_definition |
    check_constraint_definition

SQL_unique_constraint_definition ::=
    (see 11.7 in ISO/IEC 9075:1992)

SQL_referential_constraint_definition ::=
    (see 11.8 in ISO/IEC 9075:1992)

check_constraint_definition ::=
    check ( search_condition )
```

1. If present, identifier_2 shall be equivalent to identifier_1. Identifier_1 is the *name* of the table and the table_definition.

2. The name of the table_definition shall be different from the name of any other table_definition or view_definition in the enclosing schema_module.

3. A table_definition shall contain at least one column_definition.

4. Every *SQL*_column_name shall be distinct from every other *SQL*_column_name within the enclosing table_definition.

5. If the column_constraint is absent from a column_definition, then the domain_reference shall not be to a not null only domain.

6. For the semantics of **not null**, see 11.4 and 11.7 of ISO/IEC 9075:1992. For the semantics of **check**, see 11.4 and 11.8 of ISO/IEC 9075:1992.

7. Suppose that column_definition *CD* is of the form

    ```
    CN [DT] [DC] [CC] : D;
    ```

    a. The domain of *CD*, denoted DOMAIN(*CD*), is *D*. If *DT* is present, then conversion between DBMS_TYPE(*D*) (see 7.1.3 of this specification) and *DT* shall be legal in both directions by the rules of SQL (9.1 and 9.2 of ISO/IEC 9075:1992) unless either *DT* or DBMS_TYPE(*D*) is an implementation defined dbms_type (see 7.1.1.3 of this specification), in which case both conversions shall be legal by the implementation defined rules.

    b. Define DATACLASS(*CD*) as DATACLASS(*D*).

    c. Define LENGTH(*CD*) as LENGTH(*D*).

    d. Define SCALE(*CD*) as SCALE(*D*). See 7.1.3 of this specification.

8. If **extended** appears in a table_definition then **extended** shall also appear in the associated schema_module declaration.

9. If an extended_table_element appears in a table_definition, then the keyword **extended** shall appear in that table_definition (see 6.7 of this specification).

## 7.2.2 View Definitions

```
view_definition ::=
    [extended]view identifier_1
        [view_column_list]
        as query_spec
        [with check option]
    end [identifier_2];

query_spec ::=
    query_specification |
    extended_query_specification

view_column_list ::=
    ( column_name {, column_name } )
```

1. If present, identifier_2 shall be equivalent to identifier_1. Identifier_1 is the name of the view and the view_definition.

2. The name of the view_definition shall be different from the name of any other table_definition or view_definition in the enclosing schema module.

3. A query_spec may be an extended_query_specification only if the keyword **extended** appears in the associated view_definition.

*Note:* **Examples**

The following is an example of a schema module.

```
schema module Parts_Suppliers_Database is

  -- the Parts table
  table P is                           -- 7.2.1: #1 and #2

    PNO not null : Part_Numbers,       -- 7.2.1: #3 and #4
    PNAME        : Part_Names,         -- 7.2.1: #5
    COLOR        : Colors,
    WEIGHT       : Weight_In_Pounds,
    CITY         : City_Names,
    unique (PNO)

  end P;

  -- the Suppliers table
  table S is                           -- 7.2.1: # 1 and #2

    SNO not null : Supplier_Numbers,   -- 7.2.1: #3 and #4
    SNAME        : Supplier_Names,     -- 7.2.1: #5
    "STATUS"     : Status_Values,      -- 5.3
    CITY         : City_Names,
    unique (SNO)

  end S;

  -- the Orders table
  table SP is                              -- 7.2.1: #1 and #2

    SNO character(5) not null : Supplier_Numbers,      -- 7.2.1: #3 and #4
    PNO character(6) not null : Part_Numbers,
```

```
    QTY integer             : Quantities, -- 7.2.1: #5
    unique (SNO, PNO)

  end SP;

  -- the Part_Number_City view
  view PNO_CITY as                    -- 7.2.2: #1 and #2

    select distinct PNO, CITY
    from SP, S
      where SP.SNO = S.SNO

  end PNO_CITY;

end Parts_Suppliers_Database;
```

**End Examples**

## 7.3 Data Conversions

The procedures that are described in an abstract module (see clause 8 of this specification) transmit data between an Ada application and a DBMS. Those data undergo a conversion during the execution of those procedures. Constants and enumeration literals used in statements are replaced by their database representation in the form of the statement in the concrete module. This process occurs at module compile time. Both processes are described in this clause.

### Execution Time Conversions

The execution time conversions check for and appropriately translate null values; for not null values, the conversion method identified by the appropriate base domain definition (see 7.1.1.3 of this specification) is executed.

**Input parameter conversion rule.** If the type of an input parameter is null-bearing, then in the corresponding SQL procedure there is an associated *SQL*_parameter_specification to which an *SQL*_indicator_parameter has been assigned. (See 8.6 and 8.8 of this specification.) If, for any execution of the procedure, the value of the input parameter is null, then the indicator parameter is assigned a negative value. (See 4.17.2 and General Rule 6 in 6.2 of ISO/IEC 9075:1992.) Otherwise, the indicator parameter shall be non-negative and the SQL parameter shall be set from the input parameter by the conversion process identified for the base domain. If the type of an input parameter is not null-bearing, the SQL parameter shall be set from the input parameter by the conversion process identified for the base domain (7.1.1.3 of this specification).

**Output parameter conversion rule.** For output parameters of procedures containing either **fetch** or **select** statements, this process is run in reverse. Let SP be a select parameter. Then the corresponding SQL procedure has a data parameter and an indicator parameter corresponding to SP (see 8.2, 8.5, and 8.7 of this specification). For any execution of the procedure:

-- If the indicator parameter is negative, then

  -- if the type of the Ada record component $COMP_{Ada}$(SP) (see 8.2 and 8.5 of this specification) is null-bearing, then $COMP_{Ada}$(SP) is set to the null value; *else*

  -- if the type of $COMP_{Ada}$(SP) is not null-bearing, the exception SAMeDL_Standard.Null_Value_Error is raised.

-- If the indicator parameter is non-negative, then the value of $COMP_{Ada}$(SP) is set from the value of the SQL data parameter by the conversion process identified for the base domain of SP (see 7.1.1.3) of this specification). If the record component $DBLeng_{Ada}$(SP) is present (see 8.2 and 8.4 of this specification), then it is set to the value of the indicator parameter.

### Compile Time Conversions

The SQL semantics of constants, domain parameters, and enumeration literals (and constants that evaluate to enumeration literals) used in value lists of insert statements (see 8.8) and value expressions (see 8.10 of this specification) require that they be replaced in the generated SQL code by representations known to the DBMS. For enumeration literals, the enumeration mapping is used (see 7.1.1.1, 7.1.1.3 and 7.1.3 of this specification).

Let *V* be an identifier. If *V* is not a reference to a constant, a domain parameter, or an enumeration literal, then *V* is not static and undergoes no compile time conversion.

If *V* is a reference to

-- a constant declared by

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

-- a domain parameter *param* of domain D, and D assigns the expression E to *param* (see 7.1.3 of this specification),

-- or an enumeration literal *El* from enumeration domain D (see 8.3, 8.8, 8.10, and 8.11 of this specification), and D assigns the expression E to *V*,

then *V* is replaced by the static expression $SQL_{VE}$(E) (see 8.10 of this specification).

# 8 Abstract Module Description Language

## 8.1 Abstract Modules

```
abstract_module ::=
    [context]
    [extended]abstract module identifier_1 is
        authorization schema_reference
        {definition}
        {procedure_or_cursor}
    end [identifier_2];

procedure_or_cursor ::=
    cursor_declaration | procedure_declaration
```

1. When present, identifier_2 shall be equivalent to identifier_1. Identifier_1 is the *name* of the abstract module.

2. No two of the items (that is, procedures, cursors, and definitions) declared within an abstract module shall have the same name.

3. For the meaning of "**authorization** schema_reference," see 12.1 of ISO/IEC 9075:1992.

4. A procedure_or_cursor may be an extended procedure or an extended cursor (see 8.2 and 8.4 of this specification) only if the keyword **extended** appears in the abstract module declaration.

### Ada Semantics

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

### SQL Semantics

There is an SQL module associated with each abstract module that gives the SQL semantics of the abstract module. The name of the SQL module is implementation defined. The language clause of the SQL module shall specify Ada. The module authorization clause is implementation defined. See clause 12 of ISO/IEC 9075:1992.

## 8.2 Procedures

The procedures discussed in this clause are not associated with a cursor. Cursor procedures are discussed in 8.5 of this specification.

For every procedure declared within an abstract module there is an Ada procedure declared within the library unit package specification corresponding to that abstract module (see 8.1 of this specification) and an SQL procedure declared within the corresponding SQL module. A call to the Ada procedure results in the execution of the SQL procedure.

```
procedure_declaration ::=
    [extended] procedure identifier_1 [input_parameter_list] is
        statement [status_clause];

statement ::=
        commit_statement | delete_statement |
        insert_statement_values | insert_statement_query |
        rollback_statement |
        select_statement | update_statement |
        extended_statement
```

1. Identifier_1 is the *name* of the procedure.

2. An input_parameter_list may appear only in conjunction with statements that take input parameters or with extended statements. In particular, such lists shall not appear in conjunction with a commit_statement, rollback_statement, or an insert_statement_values.

3. A statement may be an extended_statement only if the keyword **extended** appears in the procedure declaration. If the keyword **extended** appears in the procedure declaration, then the keyword **extended** shall appear within the definition of the module in which the procedure is declared.

### Ada Semantics

Each procedure declaration P shall be assigned an Ada procedure declaration $P_{Ada}$ in a manner that satisfies the following constraints:

-- If P is declared within the declaration of an abstract module M, then $P_{Ada}$ is declared directly within the library unit package specification M.

-- The simple name of $P_{Ada}$ is AdaID(Identifier_1), the name of P.

The parameter profile of the Ada procedure is defined as follows:

1. If the statement within the procedure is either a delete, insert_statement_query, select or update statement, then let there be $k$ input parameters (for some $k \geq 0$) in the input_parameter_list given by $INP_1$, $INP_2$, . . . , $INP_k$. Then the i[th] parameter declaration in the *Ada*_formal_part of $P_{Ada}$, denoted $PARM_{Ada}(INP_i)$ for $i \leq k$, is given by

   AdaNAME($INP_i$) : **in** AdaTYPE($INP_i$)

   (See 8.6 of this specification.)

2. If the statement within the procedure is a select_statement, then the $(k+1)^{st}$ parameter in the *Ada*_formal_part of $P_{Ada}$ is a row record. The mode of the row record parameter shall be **in out**.

   Let *IC* be the into_clause appearing (possibly by assumption, see 8.3 of this specification) in the select_statement. Then the name of the row record parameter is $PARM_{Row}$(*IC*); the name of the type of that parameter is $TYPE_{Row}$(*IC*). See 8.9 of this specification. If *IC* contains the keyword **new**, then the declarative region containing the declaration of $P_{Ada}$ shall also contain the declaration of $TYPE_{Row}$(*IC*).

   The names, types, and order of the components of the row record parameter are determined from the select_list within the select_statement. Let that list be given by $SP_1$; $SP_2$; . . . ; $SP_m$. (If the select_list takes the form '$*$' then assume the transformation described in 8.7 of this specification has been applied.) Then the row record type is equivalent in the sense of 3.2.10 and 3.7.2 of ISO 8652:1987, to a record whose sequence of components is given by the sequence

   ... $COMP_{Ada}(SP_i)$ [$DBLeng_{Ada}(SP_i)$] ...

   where $COMP_{Ada}(SP_i)$ is given by

   AdaNAME($SP_i$) : AdaTYPE($SP_i$)

   *provided* that AdaNAME($SP_i$) and AdaTYPE($SP_i$) are defined (see 8.7 of this specification). The record component $COMP_{Ada}(SP_i)$ is otherwise undefined. The record component $DBLeng_{Ada}(SP_i)$ is given by

   DBLngNAME($SP_i$) : Ada_Indicator_Type

   where Ada_Indicator_Type is the type SQL_Standard.Indicator_Type (see 12.3.8.a.iii of ISO/IEC 9075:1992), *provided* that DBLngNAME($SP_i$) is defined; otherwise this component is not present.

   *Note*: $COMP_{Ada}(SP_i)$ is undefined only if the i[th] select parameter is improperly written; whereas $DBLeng_{Ada}(SP_i)$ is undefined when the i[th] select parameter does not have a **dblength** phrase. See 8.7 of this specification.

3. If the statement within the procedure is an insert_statement_values and it is *not* the case that the insert_value_list is present and consists solely of literals and constants, then the first parameter is a row record. The mode of the record parameter is **in**.

   Let *IC* be the insert_from_clause appearing (possibly by assumption, see 8.3 of this specification) in the statement. Then the name of the row record parameter is $PARM_{Row}$(*IC*); the name of the type of that parameter is $TYPE_{Row}$(*IC*). See 8.9 of this specification. If *IC* contains the keyword **new**, then the declarative region containing the declaration of $P_{Ada}$ also contains the declaration of $TYPE_{Row}$(*IC*).

   The names, types, and order of the components of the record type are determined from the insert_column_list and insert_value_list. So, let $C_1$, $C_2$, . . . , $C_m$ be the subsequence of insert_column_specifications appearing in an insert_column_list such that the corresponding element of the insert_value_list is not a literal or constant reference. Then the row record type is equivalent in the sense of 3.2.10 and 3.7.2 of ISO 8652:1987, to the record whose $i^{th}$ component $COMP_{Ada}(C_i)$ for $1 \le i \le m$ is given by

```
AdaNAME(Ci) : AdaTYPE(Ci)
```

(See 8.8 of this specification).

4. If the statement within the procedure is an extended_statement, see 6.7 of this specification; for extended parameter lists, see 8.6 of this specification.

5. For all procedures, regardless of statement type, if a status_clause appears in the procedure declaration, then the final parameter is a status parameter of mode **out**. For the name and type of this parameter, see 7.1.8 and 8.13 of this specification.

## SQL Semantics

Each procedure declaration P shall be assigned an SQL procedure $P_{SQL}$ within the SQL module for the abstract module in which the procedure appears. $P_{SQL}$ has three parts:

1. An *SQL*_procedure_name. This is implementation defined.

2. A list of *SQL*_parameter_declarations. An SQLSTATE parameter is declared for every SQL procedure unless a status_clause appears in the procedure declaration that references a status map specifying **sqlcode**, in which case an SQLCODE parameter is declared. (See 7.1.8 and 8.13 of this specification.) Other parameters depend on the type of the statement within the procedure P.

   a. If the statement is a delete, insert_statement_query, select or update statement, then the SQL parameters derived from the input_parameter_list of the procedure, as described in 8.6 of this specification, appear in the parameter declarations of $P_{SQL}$.

   b. If the statement is an insert_statement_values, then the SQL parameters are determined by the subsequence of insert_column_specifications in the insert_column_list whose corresponding entry in the insert_value_list is a column_name (thus not a literal or constant_reference). See 8.8 of this specification.

   c. If the statement is a select_statement, then the SQL parameter declarations for $P_{SQL}$ are determined by the select_list of the select_statement, as described in 8.7 of this specification.

   d. If the statement is an extended_statement, see 6.7 of this specification.

3. An *SQL*_SQL_data_statement or *SQL*_SQL_data_change_statement (see 12.5 in ISO/IEC 9075:1992). This is derived from the statement in the procedure declaration by the rules in 6.7 and 8.3 of this specification.

## Interface Semantics

A call to the Ada procedure $P_{Ada}$ shall have effects that cannot be distinguished from the following:

1. The procedure $P_{SQL}$ is executed in an environment in which the values of parameters $PARM_{SQL}(INP)$ and $INDIC_{SQL}(INP)$ (see 8.6 of this specification) are set from the value of $PARM_{Ada}(INP)$ (see Ada Semantics above) according to the rule for input parameters of 7.3 of this specification. This holds for every input parameter INP in the input_parameter_list of the procedure or for every column parameter INP in the insert_column_list of an insert_statement_values whose corresponding entry in the insert_column_list is an *SQL*_column_name (thus not a literal or constant_reference). See 8.8 of this specification.

2. Standard post processing, as described in 6.6 of this specification, is performed.

3. If the value of the SQL status parameter indicates successful completion or if it is handled by a status map attached to the procedure (see 6.6 of this specification) and is defined by the SQL-implementation to permit the transmission of data and the statement within the procedure is a select_statement, then the value of the component of the row record parameter components $COMP_{Ada}(SP_i)$ and $DBLeng_{Ada}(SP_i)$ are set from the values of the actual parameters associated with the SQL formal parameters $PARM_{SQL}(SP_i)$ and $INDIC_{SQL}(SP_i)$ (see 8.7 of this specification), according to the rule for output parameters of 7.3 of this specification.

*Note:* **Examples**

The following are examples of procedure declarations. The first is a declaration of a procedure with no input parameters.

```
procedure Parts_Suppliers_Commit is

   commit work;
```

The above declaration produces the following Ada procedure specification in the abstract interface.

```
procedure Parts_Suppliers_Commit;
```

The next procedure declaration contains an input parameter and a status clause.

```
procedure Delete_Parts (
   Input_Pname named Part_Name : Part_Names) is

   delete from P
      where PNAME = Input_Pname
      status Operation_Map named Delete_Status
;
```

The above declaration produces the following Ada procedure specification in the abstract interface.

```
procedure Delete_Parts (
   Part_Name     : in Part_Names_Type;      -- 8.6: Ada Semantics #1
                                          -- 8.2: Ada Semantics #1, #2
   Delete_Status : out Operation_Status); -- 8.2: Ada Semantics #5
```

A somewhat more complex example, involving a row record: the SAMeDL procedure:

```
procedure Insert_Redparts is
   insert into P
      (PNO named Part_Number,
       PNAME named Part_Name,
       COLOR,
       CITY)
   from Red_Parts
   values (PNO, PNAME, Red, CITY);
```

produces the following Ada declarations:

```
type Insert_Redparts_Row_Type -- 8.2, Ada semantics #3, 8.9
   is record
      Part_Number : Part_Numbers_Type;   -- 7.2.1
      Part_Name   : Part_Names_Type;
      City        : City_Names_Type;
   end record;

procedure Insert_Redparts
   (Red_Parts : in Insert_Redparts_Row_Type);
```

The color of all parts inserted using the Insert_Redparts procedure will be red. The weight of all such parts will be null. See the examples in 7.2.2 of this specification. The number, name and city of those parts are specified at run time.

**End Examples**

## 8.3 Statements

This clause describes the concrete syntax of statements other than cursor-oriented statements. The text of the SQL statement derived from the text of a statement is defined.

```
commit_statement ::= commit work

rollback_statement ::= rollback work

delete_statement ::=
    delete from table_name
    [where search_condition]

insert_statement_query ::=
    insert into table_name [(SQL_insert_column_list)]
        query_specification

insert_statement_values ::=
    insert into table_name [(insert_column_list)]
    [insert_from_clause] values [(insert_value_list)]

update_statement ::=
    update table_name
    set set_item {, set_item}
    [where search_condition]

set_item ::=
        column_reference = update_value

update_value ::=
        null | value_expression

select_statement ::=
        select [distinct | all] select_list
        [into_clause]
        from_clause
        [where search_condition]
```

In the following discussion, let *ProcName* be the name of the procedure in which the statement appears.

1. If no insert_from_clause appears within an insert_statement_values, then the following clause is assumed:

   `from Row : new ProcName_Row_Type`

   If an insert_from_clause which does not contain a record_id appears in an insert_statement_values, the record_id

   `: new ProcName_Row_Type`

   is assumed. See 8.9 of this specification.

2. If no into_clause appears within a select_statement, then the following clause is assumed:

   `into Row : new ProcName_Row_Type`

   If an into_clause which does not contain a record_id appears in a select_statement, the record_id

   `: new ProcName_Row_Type`

   is assumed. See 8.9 of this specification.

3. This rule applies to both forms of insert statements. If an insert_column_list is not present, then a column list consisting of all columns defined for the table denoted by *SQL*_table_name is assumed, in the order in which the columns were declared (8.7.3 of ISO/IEC 9075:1992).

*Note*: Use of the empty insert_column_list is considered poor programming practice. The interpretation of the empty insert_column_list is subject to change as the database design changes. Programs that use an empty insert_column_list present maintenance difficulties not presented by programs supplying an insert_column_list.

4. If the statement is an insert_statement_values, then

   a. If the insert_value_list is not present, then a list consisting of the sequence of column names in the insert_column_list is assumed.

   b. The insert_column_list and insert_value_list shall conform, as described in 8.8 of this specification.

5. If the statement is an insert_statement_query, then let $C_1$, $C_2$, . . . , $C_m$ be the columns appearing in an *SQL*_insert_column_list, and for each $1 \le i \le m$, let $D_i$ be DBMS_TYPE($C_i$) (see 7.2 of this specification). The select_parameters in the select_list of the query_specification shall not specify a named_phrase or a not null phrase; that is, the select_list shall have the form $VE_1$, $VE_2$, . . . , $VE_n$, for value_expressions, $VE_i$. Then

   -- m=n, that is, the lists are of the same length; and

   -- for each $1 \le i \le n$, $VE_i$ shall conform to $D_i$ (see 6.5 of this specification) and if DATACLASS($D_i$) is character, then LENGTH($VE_i$) shall not exceed LENGTH($D_i$).

6. The following apply to update statements. Let

   ```
   C = v
   ```

   be a set_item within an update_statement. Let D be DOMAIN(C). Then

   a. If v is **null**, D shall not be a not null only domain.

   b. Otherwise, v is a value_expression. v shall conform to D (see 6.5 of this specification) and if DATACLASS(D) is character, LENGTH(v) shall not exceed LENGTH(D).

## SQL Semantics

This clause describes the text of an SQL statement corresponding to the statement within a procedure.

1. The SQL commit and rollback statements are textually identical to the commit and rollback statements.

2. The delete_statement is transformed into an *SQL*_delete_statement_searched by applying the transformation $SQL_{SC}$ described in 8.11 of this specification to the search condition of the where clause, if present. The remainder of the statement is unchanged.

3. The insert_statement_query is transformed into an *SQL*_insert_statement by

   a. Applying the transformation $SQL_{VE}$ defined in 8.10 of this specification to the value_expression in each select_parameter of the select_list in the query_specification.

   b. Removing any **as** keywords, if present, from the from_clause in the query_specification.

   c. Applying the transformation $SQL_{SC}$ described in 8.11 of this specification to the search_conditions, if any, in the query_specification.

   The remainder of the statement is unchanged.

4. The insert_statement_values is transformed into an *SQL*_insert_statement by transforming the insert_value_list and insert_column_list as described in 8.8 of this specification, and dropping the insert_from clause, if present. The remainder of the statement is unchanged.

5. The update_statement is transformed into an *SQL*_update_statement: searched by applying the transformation $SQL_{VE}$ to the value expressions in the set_items of the statement and by applying the transformation $SQL_{SC}$ to the search condition, if present. The remainder of the statement is unchanged.

6. The select_statement is transformed into an *SQL*_select_statement: single row by

   a. Replacing the select_list with the *SQL*_select_list described in 8.7 of this specification.

   b. Inserting the phrase INTO *target list*, where *target list* is as specified in 8.7 of this specification, and removing the into_clause in the statement, if any.

   c. Removing any **as** keywords, if present, from the from_clause.

   d. Applying the transformation $SQL_{SC}$ described in 8.11 of this specification to the search conditions, if any, in the where and having clauses.

   The remainder of the statement is unchanged.

## 8.4 Cursor Declarations

```
cursor_declaration ::=
    [extended] cursor identifier_1
        [input_parameter_list]
        for
            query
            [SQL_order_by_clause]
        ;
        [ is cursor_procedures
    end [identifier_2];]

query ::=
    query_expression | extended_query_expression

query_expression ::=
    query_term |
    query_expression union [all] query_term

query_term ::=
    query_specification |
    (query_expression)

query_specification ::=
    select [distinct | all ] select_list
        from_clause
        [where search_condition]
        [SQL_group_by_clause]
        [having search_condition]
```

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

2. No two procedures within a cursor_declaration shall have the same name.

3. A query may be an extended_query_expression only if the keyword **extended** appears in the cursor declaration. If the keyword **extended** appears in the cursor declaration, then the keyword **extended** shall appear in the declaration of the module in which the cursor is declared (see 6.2 of this specification).

## Ada Semantics

If a cursor named *C* is declared within an abstract module named *M*, then a subpackage named *C* shall exist within the Ada package *M* (see 8.1 of this specification). That subpackage shall contain the declarations of the procedures declared in the sequence cursor_procedures. (Some of those procedures may appear by assumption. See 8.5 of this specification.) The text of the procedure declarations is described in 8.5 of this specification.

If the query in a cursor declaration is an extended_query_expression, see 6.7 of this specification.

If there is no **union** operator in the query_expression in the cursor_declaration, then the names, types, and order of the components of any record type used as a row record formal parameter type in any fetch procedure for this cursor are determined from the select_list as specified for the select_statement in 8.2 and 8.7 of this specification. Otherwise, if **union** is present, the select_lists of all the query_expressions in the cursor_declaration shall have the same length. The name and type of the $i^{th}$ component of the record type is determined by the set of select_parameters in the $i^{th}$ location of the select_lists. Let there be *m* such select_lists and let the set of select_parameters appearing in the $i^{th}$ location of these lists be denoted by $\{SP^j_i\} = \{VE^j_i$ [**named** $Id^j_i$] [**not null**$^j_i$]$\}$ $1 \le j \le m$. Then

1. All these parameters have the same Ada type, that is, $AdaTYPE(SP^j_i) = AdaTYPE(SP^k_i)$ for all pairs $1 \le j,k \le m$ (see 8.7 of this specification). The Ada type of the $i^{th}$ parameter, $AdaTYPE_i$, is that type; in other words, $AdaTYPE_i = AdaTYPE(SP^j_i)$ for any $1 \le j \le m$.

   *Note*: This is equivalent to the restriction that $DOMAIN(VE^j_i)$ is the same domain, say $DOMAIN_i$, for all values of j (see 8.10 of this specification) and that either (i) $DOMAIN_i$ is a not null only domain, or (ii) **not null** is specified for either all or none of the parameters.

2. For all pairs j, k such that a named_phrase appears in $SP^j_i$ and $SP^k_i$, $Id^j_i$ shall equal $Id^k_i$. Then that name, $AdaNAME_i$, satisfies $AdaNAME_i = Id^j_i$ for any such j. If there are no such pairs (that is, if a **named** phrase appears in none of the select_parameters), then $AdaNAME(VE^j_i)$ shall equal $AdaNAME(VE^k_i)$ for all pairs $1 \le j,k \le m$ and shall not equal NO_NAME (see 8.10 of this specification). Then $AdaNAME_i = AdaNAME(VE^j_i)$ for any $j \le m$.

3. For all pairs j, k such that a **dblength** phrase appears in either $SP^j_i$ or $SP^k_i$, then a **dblength** phrase shall appear in both $SP^j_i$ and $SP^k_i$. Furthermore, $DBLngNAME(SP^j_i)$ shall equal $DBLngNAME(SP^k_i)$. Then $DBLngNAME_i$ shall be that name. If the **dblength** phrase appears in no $SP^j_i$ for any j, then $DBLngNAME_i$ is said to be null; otherwise, $DBLngNAME_i$ is undefined.

The type of the row record parameter is equivalent in the sense of 3.2.10 and 3.7.2 of ISO 8652:1987, to a record type whose sequence of components is given by the sequence

$$\ldots\ COMP_{Ada}(SP_i)\ [DBLeng_{Ada}(SP_i)]\ \ldots$$

where $COMP_{Ada}(SP_i)$ is given by

$$AdaNAME_i : AdaTYPE_i$$

*provided* that $AdaNAME_i$ and $AdaTYPE_i$ are defined; the record component $COMP_{Ada}(SP_i)$ is otherwise undefined. The record component $DBLeng_{Ada}(SP_i)$ is given by

$$DBLngNAME_i : Ada\_Indicator\_Type$$

where Ada_Indicator_Type is the type SQL_Standard.Indicator_Type (see 12.3.8.a.iii in ISO/IEC 9075:1992), *provided* that $DBLngNAME_i$ is defined. If $DBLngNAME_i$ is null, this component is not present in the row record parameter.

## SQL Semantics

A cursor declaration is transformed into an SQL cursor declaration as follows.

1. The string "declare " is prepended to the cursor declaration and, if present, the keyword **extended** is dropped.
2. The input_parameter_list and cursor_procedures are discarded, as is the keyword **cursor** and the **is** . . . **end** bracket. The cursor name identifier_1 is transformed into $SQL_{NAME}$(identifier_1).
3. The string " cursor " is inserted immediately after the transformed cursor name, but before the keyword **for**.
4. The select_list is transformed into an *SQL*_select_list as described in 8.7 of this specification.
5. Any **as** keywords present are removed from the from_clause.
6. The search conditions are transformed using the transform $SQL_{SC}$ of 8.11 of this specification.

The remainder of the declaration is unchanged.

*Note:* **Examples**

The following note consists of two examples of cursor_declarations: the first contains a simple cursor_declaration, while the second contains a more complex declaration that exercises many of the features of the syntax. In both cases the generated Ada code is shown, annotated with references to the appropriate clauses of the language definition.

The example below is a simple cursor_declaration.

```
cursor Select_Suppliers
  for
    select SNO, SNAME, "STATUS", CITY dblength  -- 5.3
      from S
  ;
```

This declaration produces the following Ada code.

```
package Select_Suppliers is              -- 8.4: Ada Semantics

  type Row_Type is record                -- 8.5, #5 and #8
    Sno     : Supplier_Numbers_Type;     -- 8.5, #3 and #8
    Sname   : Supplier_Names_Type;
    STATUS  : Status_Values_Type;
    City    : City_Names_Type;
    City_Dblength : SQL_Standard.Indicator_Type;
  end record;

  procedure Open;      -- 8.5: #3

  procedure Fetch (    -- 8.5: #5
    Row      : in out Row_Type;          -- 8.5: #8 and Ada Semantics #3
    Is_Found : out boolean);             -- 8.5: #5 and Ada Semantics #6

  procedure Close;     -- 8.5: #4

end Select_Suppliers;
```

The following is an example of a more complex cursor_declaration.

```
cursor Supplier_Operations (
  Input_City named Supplier_City     : City_Names not null;
  Adjustment named Status_Adjustment : Status_Values not null)

  for
    select SNO named Supplier_Number,
           SNAME named Supplier_Name,
```

```
            "STATUS" + Adjustment named Adjusted_Status,  -- 5.3
            CITY named Supplier_City
        from S
        where CITY = Input_City
    ;

    is

      procedure Open_Supplier_Operations is
        open Supplier_Operations;

      procedure Fetch_Supplier_Tuple is
        fetch Supplier_Operations
          into Supplier_Row_Record : new Supplier_Row_Record_Type
          status My_Map named Fetch_Status;

      procedure Close_Supplier_Operations is
        close; -- optional 'cursor name' omitted

      procedure Update_Supplier_Status (
        Input_Status named Updated_Status : Status_Values not null;
        Input_Adjustment named Adjustment : Status_Values) is

        update S
          set "STATUS" = Input_Status + Input_Adjustment  - 5.3
          where current of Supplier_Operations;

      procedure Delete_Supplier is
        delete from S;
        -- optional "where current of 'cursor name'" omitted

  end Supplier_Operations;
```

This declaration produces the following Ada code.

```
  package Supplier_Operations is       -- 8.4: Ada Semantics

    type Supplier_Row_Record_Type is
      record                           -- 8.5: Ada Semantics #5 #8
        Supplier_Number : Supplier_Numbers_Type;
                                       -- 8.5: Ada Semantics #3 #8
        Supplier_Name   : Supplier_Names_Type;
        Adjusted_Status : Status_Values_Type;
        Supplier_City   : City_Names_Type;
      end record;

      procedure Open_Supplier_Operations (
        Supplier_City     : in City_Names_Not_Null;  -- 8.5: Ada Semantics
        Status_Adjustment : in Status_Values_Not_Null); --   #1, #3, Modes

      procedure Fetch_Supplier_Tuple (
        Supplier_Row_Record : in out Supplier_Row_Record_Type;  -- 8.5
        Fetch_Status        : out Operation_Status);            -- 8.5

      procedure Close_Supplier_Operations;

      procedure Update_Supplier_Status (
        Updated_Status : in Status_Values_Not_Null; -- 8.5: Ada Semantics #2
        Adjustment     : in Status_Values_Type);    -- 8.5: Ada Semantics

      procedure Delete_Supplier;

  end Supplier_Operations;
```

**End Examples**

## 8.5 Cursor Procedures

```
cursor_procedures ::=
    cursor_procedure {cursor_procedure}

cursor_procedure ::=
    [extended] procedure identifier_1
    [input_parameter_list] is
    cursor_statement
    [status_clause]
    ;

cursor_statement ::=
    open_statement | fetch_statement | close_statement |
    cursor_update_statement | cursor_delete_statement |
    extended_cursor_statement

open_statement ::=
    open [identifier]

fetch_statement ::=
    fetch [identifier] [into_clause]

close_statement ::=
    close [identifier]

cursor_update_statement ::=
    update table_name
    set set_item {, set_item}
    [where current of identifier]

cursor_delete_statement ::=
    delete from table_name
    [where current of identifier]
```

1. Identifier_1 is the *name* of the procedure.

2. An input parameter list may only appear in conjunction with statements that take input parameters. In particular, such lists shall not appear in conjunction with open, close, fetch and cursor_delete statements. Only a cursor_update_statement or an extended_cursor_statement may take an input_parameter_list.

3. If no open_statement appears in a list of cursor_procedures, the declaration **procedure** "open" **is open**; is assumed.

4. If no close_statement appears in a list of cursor_procedures, the declaration **procedure** "close" **is close**; is assumed.

5. If no fetch_statement appears in a list of cursor_procedures, the declaration **procedure** "fetch" **is fetch status standard_map**; is assumed. See 7.1.8 of this specification.

6. If identifier is present in an open, fetch, close, cursor_update or cursor_delete statement, then it shall be equal to the name of the cursor within which the procedure declaration appears. The meaning of a cursor statement is not affected by the presence or absence of these identifiers.

7. The restrictions that apply to the set_items of a non-cursor update_statement (see 8.3 of this specification) also apply to the set_items of a cursor_update_statement.

8. If no into_clause appears within a fetch_statement, then the following clause is assumed:

```
into Row : new Row_Type
```

If an into_clause which does not contain a record_id appears in a fetch_statement, the record_id

```
: new Row_Type
```

is assumed. See 8.9 of this specification.

9. A cursor_statement may be an extended_cursor_statement only if the keyword **extended** appears in the cursor_procedure declaration. If the keyword **extended** appears in the cursor_procedure declaration, then the keyword **extended** shall appear within the declaration of the cursor in which the cursor_procedure is declared.

## Ada Semantics

Each procedure declaration P that appears in or is assumed to appear in a cursor_procedures list shall be assigned an Ada procedure declaration $P_{Ada}$ that satisfies the following constraints.

-- If P is declared within the declaration of a cursor named *C*, then $P_{Ada}$ shall be declared within the specification of an Ada subpackage named *C*.

-- The simple name of $P_{Ada}$ is AdaID(identifier_1), the name of P.

*Note*: The default open, fetch and close procedures use delimited identifiers (see 5.3 of this specification) as their procedure names.

The parameter profiles of the Ada procedures depend in part on the statement within the procedure, as follows:

1. For open_statements: Let $INP_1$, $INP_2$, . . . , $INP_k$ $k \geq 0$ be the list of input parameters in the input_parameter_list of the cursor_declaration within which the procedure appears. Then $PARM_{Ada}(INP_i)$, the $i^{th}$ parameter of the *Ada*_formal_part, is of the form

   AdaNAME($INP_i$) : **in** AdaTYPE($INP_i$)

   for $1 \leq i \leq k$ (see 8.6 of this specification).

2. For cursor_update_statements: Let $INP_1$, $INP_2$, . . . , $INP_k$ $k \geq 0$ be the list of input parameters in the input_parameter_list of the statement. Then $PARM_{Ada}(INP_i)$, the $i^{th}$ parameter of the *Ada*_formal_part, is of the form

   AdaNAME($INP_i$) : **in** AdaTYPE($INP_i$)

   for $1 \leq i \leq k$ (see 8.6 of this specification).

3. For fetch_statements: The first parameter is a row record parameter of mode **in out**. The names, order, and types of the components of the type of this parameter are described in 8.2 and 8.4 of this specification. Let *IC* be the into_clause appearing (possibly by assumption) in the fetch_statement. Then the name of the row record formal parameter is $PARM_{Row}(IC)$, and the type of that parameter is $TYPE_{Row}(IC)$. See 8.9 of this specification. If *IC* contains the keyword **new**, then the declarative region containing the declaration of $P_{Ada}$ shall contain the declaration of $TYPE_{Row}(IC)$.

4. For close and cursor_delete statements: There are no parameters to these procedures (except possibly for a status parameter).

5. For extended_cursor_statements, see 6.7 of this specification. For extended parameter lists, see 8.6 of this specification.

6. For all statement types: If a status_clause referencing a status map that contains a **uses** appears in the procedure declaration, then the final parameter is a status parameter of mode **out**. For the name and type of this parameter, see 7.1.8 and 8.13 of this specification.

## SQL_Semantics

Each procedure P that appears in or is assumed to appear in a cursor_procedures list shall be assigned an SQL procedure $P_{SQL}$ within the SQL module for the abstract module within which the cursor_procedures list appears. $P_{SQL}$ has three parts:

1. An *SQL*_procedure_name. This is implementation defined.

2. A list of *SQL*_parameter_declarations. An SQLSTATE parameter is declared for every SQL procedure unless a status clause appears in the procedure declaration that references a status map specifying **sqlcode**, in which case an SQLCODE parameter is declared. (See 7.1.8 and 8.13 of this specification.) Other parameters depend on the type of the statement within the procedure P.

   a. If the statement is an open_statement, then the SQL parameters derived from the input_parameter_list of the cursor_declaration as described in 8.6 of this specification appear in the parameter declarations of $P_{SQL}$.

   b. If the statement is a cursor_update_statement, then the SQL parameters derived from the input_parameter_list of the cursor_update_statement as described in 8.6 of this specification appear in the parameter declarations of $P_{SQL}$.

   c. If the statement is a fetch_statement, then the SQL parameters determined by the select_list of the cursor_declaration (as described in 8.7) of this specification appear in the parameter declarations of $P_{SQL}$.

   The order of the parameters within the list is implementation defined.

3. An *SQL*_SQL_data_statement or *SQL*_SQL_data_change_statement. (See 12. in ISO/IEC 9075:1992.) This is derived from the statement in the procedure declaration, as follows.

   a. If the statement is an open_statement, then the *SQL*_open_statement is **open** $SQL_{NAME}$(*C*), where *C* is the cursor name.

   b. If the statement is a close_statement, then the *SQL*_close_statement is **close** $SQL_{NAME}$(*C*), where *C* is the cursor name.

   c. If the statement is the cursor_delete_statement

      **delete from** *Id_1* [**where current of** *C*]

      then the *SQL*_delete_statement_positioned is identical, up to the addition of the where phrase, **where current of** $SQL_{NAME}$(C), replacing the where phrase of the cursor_delete_statement, if present.

   d. If the statement is the cursor_update_statement

      **update** *Id_1*
      **set** *set_items*
      [**where current of** *C*]

      then the *SQL*_update_statement_positioned is formed by applying the transformation $SQL_{VE}$ defined in 8.10 of this specification to the value expressions in the set_items of the statement and appending or replacing the where phrase so as to read **where current of** $SQL_{NAME}$(C).

e. If the statement is a fetch_statement, then the *SQL*_fetch_statement is

**fetch** $\texttt{SQL}_{\texttt{NAME}}$(*C*) **into** *target list*

where *C* is the cursor name and *target list* as described in 8.7 of this specification.

f. If the statement is an extended_statement, see 6.7 of this specification.

### Interface Semantics

A call to the Ada procedure $P_{Ada}$ shall have effects that can not be distinguished from the following:

1. The procedure $P_{SQL}$ is executed in an environment in which the values of parameters $PARM_{SQL}$(INP) and $INDIC_{SQL}$(INP) (see 8.6 of this specification) are set from the value of $PARM_{Ada}$(INP) (see Ada semantics above) according to the rule of 7.3 of this specification for every input parameter, INP, in either the input_parameter_list of the cursor_declaration for open procedures, or the input_parameter_list of the procedure itself for update procedures.

2. Standard post processing, as described in 6.6 of this specification, is performed.

3. If the value of the SQL status parameter indicates successful completion or if it is handled by a status map attached to the procedure (see 6.6 of this specification) and is defined by the SQL-implementation to permit the transmission of data and the statement within the procedure is a fetch_statement, then the value of the row record components $COMP_{Ada}(SP_i)$ and $DBLeng_{Ada}(SP_i)$, are set from the values of the actual parameters associated with the SQL formal parameters $PARM_{SQL}(SP_i)$ and $INDIC_{SQL}(SP_i)$ (see 8.7 of this specification) according to the rule of 7.3 of this specification.

## 8.6 Input Parameter Lists

Input parameter lists declare the parameters of the procedure, cursor, or cursor procedure declaration in which they appear. The list consists of parameter declarations that are separated with semicolons, in the manner of Ada formal parameter declarations.

Each parameter declaration of a procedure or cursor procedure P is represented as an *Ada*_parameter_specification within the *Ada*_formal_part of the procedure $P_{Ada}$; each parameter declaration within a cursor declaration is represented as an *Ada*_parameter_specification within the *Ada*_formal_part of the Ada open procedure. The parameter is also represented as either one or two *SQL*_parameter_declarations within the *SQL*_procedure $P_{SQL}$. The second SQL parameter declaration, if present, declares the indicator variable for the parameter (see 4.17.2 of ISO/IEC 9075:1992).

The order of parameter specification within the *Ada*_formal_part is given by the order within the input_parameter_list. The order of the *SQL*_parameter_declarations within the list of declarations in the SQL procedure is implementation defined.

```
input_parameter_list ::=
    (parameter {; parameter})

parameter ::=
    identifier_1 [named_phrase] :
            [in] [out] domain_reference [not null]
```

## Ada Semantics

Let INP be a parameter the textual representation of which is given by

*Id_1* [**named** *Id_2*] : [**in**] [**out**] [*Id_3*.]*Id_4* [**not null**]

then *Id_1* is the *name* of the parameter.

The domain associated with INP, denoted DOMAIN(INP), is the domain referenced by [*Id_3*.]*Id_4*. Let DOMAIN(INP) = *D*. Then

-- LENGTH(INP) = LENGTH(*D*)

-- SCALE(INP) = SCALE(*D*)

-- DATACLASS(INP) = DATACLASS(*D*)

The functions AdaNAME and AdaTYPE are defined on parameters as follows:

1. If *Id_2* is present in the definition of INP, then AdaNAME(*INP*) = AdaID(*Id_2*) otherwise, AdaNAME(*INP*) = AdaID(*Id_1*). For no two parameters, $INP_1$ and $INP_2$, in an input parameter list shall AdaNAME($INP_1$) = AdaNAME($INP_2$).

2. AdaTYPE(*INP*) shall be the name of a type within the domain identified by the domain_reference [*Id_3*.]*Id_4*. If **not null** appears within the textual representation of INP, *or* the domain identified by the domain_reference is not null only, then AdaTYPE(*INP*) shall be the name of the not null-bearing type within the identified domain; otherwise it shall be the name of the null-bearing type within that domain (see 7.1.3 of this specification).

The optional **out** may occur only in a parameter that is associated with a procedure or cursor that is extended. The optional **in**, however, may be included in any parameter declaration.

Given *INP* as defined above, define mode(*INP*) to be

-- *in*, if INP either contains (1) the optional **in**, but not the optional **out**, or (2) neither **in** nor **out**.

-- *out*, if *INP* contains **out** but not **in**.

-- *in out*, if *INP* contains both **in** and **out**.

Then the generated parameter, $\mathrm{PARM}_{\mathrm{Ada}}$(INP), in the *Ada*_formal_part is of the form

AdaNAME(*INP*) : mode(*INP*) AdaTYPE(*INP*);

## SQL Semantics

Let INP be as given above and let *D* be the domain referenced by [*Id_3*.]*Id_4*. The SQL parameter $\mathrm{PARM}_{\mathrm{SQL}}$(INP) is declared by the following *SQL*_parameter_declaration

:$\mathrm{SQL}_{\mathrm{NAME}}$(*Id_1*) DBMS_TYPE(*D*)

where DBMS_TYPE(*D*) is given in 7.1.3 of this specification. If **not null** does *not* appear within the textual representation of INP, *and* [*Id_3*.]*Id_4* does *not* identify a not null only domain, then the parameter $\mathrm{INDIC}_{\mathrm{SQL}}$(INP) is defined and is declared by the *SQL*_parameter_declaration

:$\mathrm{INDIC}_{\mathrm{NAME}}$(INP) *indicator_type*

where *indicator_type* is the implementation-defined type of indicator parameters (Syntax Rule 1 of 6.2 of

ISO/IEC 9075:1992). The name $INDIC_{NAME}$(INP) shall not appear as the name of any other parameter of the enclosing procedure.

## 8.7 Select Parameter Lists

Select parameter lists serve to inform the DBMS of what data are to be retrieved by a select or fetch statement. They also specify the names and types of the components of a record type—the so called row record type—which appears as the type of a formal parameter of Ada procedure declarations for select and fetch statements. Further they specify the column names of viewed tables (7.2.2 of this specification).

```
select_list ::=
    * | select_parameter {, select_parameter}

select_parameter ::=
    value_expression [named_phrase] [not null]
        [dblength [named_phrase]]
```

1. The select list star (“∗”) is equivalent to a sequence of select parameters described as follows: Let $T_1$, $T_2$, . . . $T_k$ be the list of the exposed table names in the table expression **from** clause for the query specification in which the select list appears (see 7.9 of ISO/IEC 9075:1992). Let $U_i$, for $1 \le i \le k$ be defined as $S_i$ _ $V_i$ if $T_i$ is of the form $S_i$ **.** $V_i$ (i.e., $S_i$ is a schema_name, and $V_i$ is a table name); otherwise, $U_i$ is $T_i$. In other words, $U_i$ is $T_i$ with every dot “.” replaced by an underscore “_”. Let $A_{i,1}$, $A_{i,2}$, . . . , $A_{i,m_i}$ be the names of the columns of the table named $T_i$. Then the select list is given by the sequence $T_1$ **.** $A_{1,1}$ **named** $U_1$ _ $A_{1,1}$, $T_1$ **.** $A_{1,2}$ **named** $U_1$ _ $A_{1,2}$, . . . , $T_i$ **.** $A_{i,j}$ **named** $U_i$ _ $A_{i,j}$, . . . , $T_k$ **.** $A_{k,m}$ **named** $U_k$ _ $A_{k,m}$. That is, the columns are listed in the order in which they were defined (see 7.2 of this specification) within the order in which the tables were named in the from_clause.

   *Notes*: This definition differs from that given in syntax rule 3.b of 7.9 in ISO/IEC 9075:1992 in specifying that the column references are qualified by table name or correlation name. The table that is described in the cited rule of SQL has anonymous columns. The record type being described must have well-defined component names.

   Use of ∗ as a select list in an abstract module is considered poor programming practice. The interpretation of ∗ is subject to change with time as the database design changes. Programs that use ∗ as a select list present maintenance difficulties not presented by programs supplying a proper select list.

   In the following discussion, assume that a select list ‘∗’ has been replaced by its equivalent list, as described above.

2. If the keyword **dblength** is present, then value_expression shall have the dataclass **character**.

3. Let VE be the value_expression appearing in a select_parameter. DOMAIN(VE) shall not be NO_DOMAIN and VE shall conform to DOMAIN(VE).

### Ada Semantics

Let SP be a select parameter written as

VE [**named** *Id_1*] [**not null**]
  [**dblength** [**named** *Id_2*] ]

SP is assigned the Ada type name AdaTYPE(SP), the Ada name AdaNAME(SP) and the dblength name DBLngNAME(SP) as follows:

-- Let DOMAIN(VE) = *D* (see 8.10 of this specification) where *D* ≠ NO_DOMAIN. If **not null** appears in SP, *or* *D* is a not null only domain, then AdaTYPE(SP) is the name of the not null-bearing type name within the domain *D*; else AdaTYPE(SP) is the name of the null-bearing type within the domain *D*.

-- If DOMAIN(VE) = NO_DOMAIN then AdaTYPE(SP) is undefined.

-- If *Id_1* appears in SP, then AdaNAME(SP) = AdaID(*Id_1*); else AdaNAME(SP) = AdaNAME(VE) (see 8.10 of this specification).

-- If the **dblength** phrase appears in SP, then

-- if *Id_2* is present then DBLngNAME(SP)=AdaID(*Id_2*)

-- else, DBLngNAME(SP)=AdaNAME(SP)_DbLength

Otherwise, DBLngNAME(SP) is undefined.

-- DBLngNAME(SP) and AdaNAME(SP) shall not appear as either DBLngNAME($SP_i$) or as AdaNAME($SP_i$) for any other select_parameter $SP_i$ within the select_list that contains SP.

## SQL Semantics

From a select_list, three SQL fragments are derived:

1. An *SQL*_select_list within a select statement or cursor declaration.
2. A list of *SQL*_parameter_declarations.
3. Either an *SQL*_select_target_list within a select statement or an *SQL*_fetch_target_list within a fetch statement.

An *SQL*_select_list is derived from a select_list as follows:

-- The select_list ∗ becomes the *SQL*_select_list ∗.

-- Otherwise, suppose $SP_1$, $SP_2$, . . . , $SP_n$ is a select_list, where $SP_i$ is given by:

$VE_i$ [**named** *Id_1$_i$*] [**not null**]$_i$ [**dblength**$_i$ [**named** *Id_2$_i$*] ]

The *SQL*_select list, $SP'_1$, $SP'_2$, . . . , $SP'_n$ is formed by setting $SP'_i$ to $SQL_{VE}(VE_i)$.

For the purpose of defining the *SQL*_parameter_declarations and target list generated from a select_list, let $SP_1$, $SP_2$, . . . , $SP_n$, be the select_list supplied or the select_list that replaced the select_list ∗ as described above. Let each $SP_i$ be as given above. Then

-- There are two SQL parameters associated with each select_parameter, $SP_i$. They are $PARM_{SQL}(SP_i)$ and $INDIC_{SQL}(SP_i)$, where the *SQL*_parameter_declaration declaring $PARM_{SQL}(SP_i)$ is

:$SQL_{NAME}(SP_i)$ DBMS_TYPE(DOMAIN($VE_i$))

and the *SQL*_parameter_declaration declaring $INDIC_{SQL}(SP_i)$ is

:$INDIC_{NAME}(SP_i)$ *indicator_type*

where $SQL_{NAME}(SP_i)$ and $INDIC_{NAME}(SP_i)$ are *SQL*_identifiers not appearing elsewhere.

-- The target list generated from a select_list is a comma-separated list of *SQL*_target_specifications (6.2 of ISO/IEC 9075:1992). The $i^{th}$ *SQL*_target_specification in the target list is

:$SQL_{NAME}(SP_i)$ INDICATOR :$INDIC_{NAME}(SP_i)$

*Note*: All derived target specifications contain indicator parameters, irrespective of the presence or absence of a not_null phrase in the select parameter declaration.

## 8.8 Value Lists and Column Lists

```
insert_column_list ::=
    insert_column_specification {, insert_column_specification}

insert_column_specification ::=
    column_name [named_phrase] [not null]

insert_value_list ::=
    insert_value {, insert_value}

insert_value ::=
    null | constant_reference |
    literal | column_name |
    domain_parameter_reference
```

Each column_name within an insert_column_list shall specify the name of a column within the table into which insertions are to be made by the enclosing insert_statement_values. (See 8.3 of this specification. See also Syntax Rule 2 of 13.8 in ISO/IEC 9075:1992.)

Let *C* be the insert_column_specification

    Col [**named** *Id*] [**not null**]

Then AdaNAME(*C*) is defined to be AdaID(*Id*), if *Id* is present; otherwise it is AdaID(*Col*). Let DOMAIN(C) = DOMAIN(Col) = *D* be the domain assigned to the column named *Col*. If **not null** appears in *C*, or *D* is a not null only domain, then AdaTYPE(C) is the name of the not null-bearing type within the domain *D*; otherwise, AdaTYPE(C) is the null-bearing type within the domain *D*.

Let $CL \equiv C_1, \ldots, C_m$ be an insert_column_list; let $IL \equiv V_1, \ldots V_n$ be an insert_value_list. *CL* and *IL* are said to *conform* if:

1. $m=n$, that is, the length of the two lists is the same.

2. For each $1 \leq i \leq m$, if $V_i$ is

   a. **null**, then DOMAIN($C_i$) shall not be a not null only domain.

   b. A literal or a reference to either a constant or a domain parameter, then $V_i$ shall conform to DOMAIN($C_i$) (see 6.5 of this specification) and if DATACLASS(DOMAIN($C_i$)) is character, then LENGTH($V_i$) shall not exceed LENGTH(DOMAIN($C_i$)).

   c. A column_name, then $V_i$ shall be identical to the column_name in $C_i$.

### Ada Semantics

The insert_column_list and insert_value_list of an insert_statement_values together define the components of an Ada record type declaration. The names, types and order of those components are defined in 8.2 of this specification on the basis of the functions AdaNAME and AdaTYPE described above. For the name of the record type and its place of declaration, see 8.9 of this specification.

*Note*: If the insert_value_list contains only constants and literals, then the Ada procedure corresponding to the procedure containing the insert_statement_values statement of which these lists form a part does not have a row record parameter. See 8.2 of this specification.

## SQL Semantics

A set of SQL parameter declarations is defined from the pair of insert_column_list and insert_value_list. So again let $C_1$, . . . , $C_k$ be the subsequence of the insert_column_list such that the insert_value_list item corresponding to each $C_i$ is a column_name (and therefore neither a literal nor a constant nor a domain parameter reference). Further, let $C_i$ be represented by the text string

$\mathtt{Col}_\mathtt{i}$ [**named** $Id_i$] [**not null**$_\mathtt{i}$]

Then the SQL parameter declarations $\text{PARM}_{\text{SQL}}(C_i)$ for $1 \le i \le k$ given by

$:\mathtt{SQL}_{\text{NAME}}(\mathtt{Col}_\mathtt{i})\ \mathtt{DBMS\_TYPE(DOMAIN(Col}_\mathtt{i}\mathtt{))}$

appear in the list of SQL parameter declarations, where

1. $\text{SQL}_{\text{NAME}}(Col_i)$ is an implementation-defined identifier that appears nowhere else.

2. DBMS_TYPE(DOMAIN($\text{Col}_i$)) is as defined in 7.1.3 of this specification.

If **not null** does *not* appear in $C_i$ *and* the domain DOMAIN($\text{Col}_i$) is *not* not null only, then the parameter $\text{INDIC}_{\text{SQL}}(C_i)$ is defined and the parameter declaration

$:\mathtt{INDIC}_{\text{NAME}}(C_\mathtt{i})$ *indicator_type*

also appears in the list of SQL parameter declarations, where

1. $\text{INDIC}_{\text{NAME}}(C_i)$ is an implementation-defined identifier that appears nowhere else.

2. *indicator_type* is the implementation-defined type of indicator parameters (6.2 in ISO/IEC 9075:1992).

An insert_column_list and insert_value_list pair are transformed into an *SQL*_insert_column_list and *SQL*_table_value_constructor pair (see 7.2 and 13.8 of ISO/IEC 9075:1992) as follows:

1. An insert_column_list is transformed into an *SQL*_insert_column_list by the removal of all named_phrase and not_null phrases that appear in it.

   *Note*: This implies that the empty insert_column_list is transformed into the empty *SQL*_insert_column_list.

2. An insert_value_list is transformed into an *SQL*_table_value_constructor by replacing each list element as follows:

   a. A literal, or **null** (but excluding any enumeration literal), is replaced by itself; i.e., is unchanged.

   b. A constant_reference or enumeration literal *k* is replaced by a textual representation of its database value $\text{SQL}_{\text{VE}}(k)$ (see 7.3 and 8.10 of this specification).

   c. A column_name $Col_i$ is replaced by

      $:\mathtt{SQL}_{\text{NAME}}(Col_\mathtt{i})\ \mathtt{[INDICATOR\ :INDIC}_{\text{NAME}}(C_\mathtt{i})\mathtt{]}$

      where the INDICATOR phrase appears whenever the indicator parameter, $\text{INDIC}_{\text{SQL}}(C_i)$, is defined (see above).

   The resulting list is enclosed in parentheses and preceeded with the keyword VALUES.

## 8.9 Into_Clause and Insert_From_Clause

An into_clause is used within a select_statement or a fetch_statement, and an insert_from_clause is used within an insert_statement_values to explicitly name the row record parameter of those statements and/or the type of that parameter.

```
into_clause ::=
    into into_from_body

insert_from_clause ::=
    from into_from_body

into_from_body ::=
    identifier_1 : record_id |
    identifier_1 |
    : record_id

record_id ::=
    new identifier_2 | record_reference
```

### Ada Semantics

Define the string $\mathrm{PARM}_{\mathrm{Row}}$(*IC*) as follows, where *IC* is an into_clause or insert_from_clause.

1. If identifier_1 appears in *IC*, then $\mathrm{PARM}_{\mathrm{Row}}$(*IC*) = AdaID(identifier_1).

2. Otherwise, if record_id is a record_reference referencing the record declaration *R*, then $\mathrm{PARM}_{\mathrm{Row}}$(*IC*) = AdaNAME(*R*). See 7.1.5 of this specification.

3. Otherwise, $\mathrm{PARM}_{\mathrm{Row}}$(*IC*) = *Row*.

Define $\mathrm{TYPE}_{\mathrm{Row}}$(*IC*) as follows:

1. If record_id has the form

   **new** `identifier_2`

   then $\mathrm{TYPE}_{\mathrm{Row}}$(*IC*) = AdaID(identifier_2).

2. Otherwise, $\mathrm{TYPE}_{\mathrm{Row}}$(*IC*) is the record type referenced by the record_reference.

   *Note*: The assumptions made about into_clause and insert_from_clause in 8.3 and 8.5 of this specification are sufficient to ensure that every such clause contains a record_id, possibly by assumption. Therefore, the case of a missing record_id need not be considered in the definition of $\mathrm{TYPE}_{\mathrm{Row}}$(*IC*). If the record_id is a record_reference, then the names, types, and order of the components of the referenced record must exactly match the names, types, and order of the components of the record type declaration that would have been generated had the record_id been "**new** identifier" (see 8.4, 8.2, and 8.7 of this specification).

*Note:* **Examples**

The following is a set of example procedure declarations that illustrate various uses of **into** and **from** clauses. It is assumed that each of these procedures is declared within an abstract module, and that any enumeration, record, and status map declarations used are visible at the point at which each procedure is declared. Declarations for these constructs can be found in 7.1.6, 7.1.5, and 7.1.8 of this specification, respectively. In addition, it is assumed that the abstract modules in which these procedures are declared have direct visibility to the contents of the *Parts_Suppliers_Database* schema module shown in the examples in 7.2.2 of this specification.

The two examples below illustrate the use of a previously declared record object in the **into** clauses of select statements. The cursor and the select statement retrieve the same object, namely a part. The row record, Parts_Row_Record_Type (see 7.1.5 of this specification), contains the definition of the part abstraction.

```
cursor Parts_By_City (
  Input_City named Part_Location : City not null)

  for
    select PNO named Part_Number not null,
          PNAME named Part_Name,
          COLOR,
          Weight_In_Grams(WEIGHT * Grams_In_Pound) named Weight,
              -- 8.10, 7.1.4
          CITY
    into Parts_By_City_Row : Parts_Row_Record_Type
    from P
    where CITY = Input_City
;

procedure Parts_By_Number (
  Input_Pno named Part_Number : Pno not null) is

  select PNO named Part_Number not null,
        PNAME named Part_Name,
        COLOR,
        Weight_In_Grams(WEIGHT * Grams_In_Pound) named Weight,
            -- 8.10, 7.1.4
        CITY
  into Parts_By_Number_Row : Parts_Row_Record_Type
  from P
  where PNO = Input_Pno
  status Operation_Map named Parts_By_Number_Status
;
```

The above declarations produce the following Ada declarations at the abstract interface.

```
package Parts_By_City is

  procedure Open (
    Part_Location : in City_Names_Not_Null); -- 8.5: Ada Semantics

  procedure Fetch (
    Parts_By_City_Row : in out Parts_Row_Record_Type;
                                          -- 8.5: Ada Semantics #3
    Is_Found          : out boolean);    -- 8.5: Ada Semantics #6

  procedure Close;

end Parts_By_City;

procedure Parts_By_Number (
  Part_Number            : in Part_Numbers_Not_Null;
                           -- 8.2: Ada Semantics
  Parts_By_Number_Row    : in out Parts_Row_Record_Type;
  Parts_By_Number_Status : out Operation_Status);
```

The select procedure below illustrates the use of an **into** clause to specify the name and type of the generated row record parameter.

```
procedure Part_Name_By_Number (
  Input_Pno named Part_Number : Part_Numbers not null) is

  select PNAME named Part_Name
  into Part_Name_By_Number_Row : new Part_Name_Row_Record_Type
  from P
  where PNO = Input_Pno
  status Standard_Map
;
```

The above declaration produces the following Ada record type and procedure declarations at the abstract interface.

```
type Part_Name_Row_Record_Type is record -- 8.2: Ada Semantics
  Part_Name : Part_Names_Type;             -- 8,2
end record;

procedure Part_Name_By_Number(
```

```
    Part_Number             : in Part_Numbers_Not_Null;
    Part_Name_By_Number_Row : in out Part_Name_Row_Record_Type;
    Is_Found                : out boolean);
```

The example declaration below uses the default **from** clause, which produces a record_declaration at the abstract interface.

```
procedure Add_To_Suppliers is

  insert into S (SNO, SNAME, "STATUS", CITY)   -- 5.3
  values
  status Operation_Map named Insert_Status
;
```

The procedure_declaration above produces the following Ada code at the abstract interface.

```
type Add_To_Suppliers_Row_Type is record -- 5.3: #1
  Sno      : Supplier_Numbers_Type;    -- 5.2: Ada Semantics,
  Sname    : Supplier_Names_Type;      --    Parameter Profiles, #3
  STATUS   : Status_Values_Type;       -- 5.8: Ada Semantics
  City     : City_Names_Type;
end record;

procedure Add_To_Suppliers (
  Row           : in Add_To_Suppliers_Row_Type; -- 5.2: Ada Semantics
  Insert_Status : out Operation_Status);
```

This last example illustrates an insert values procedure declaration where all of the values are literals; therefore, no row record parameter is needed for the procedure declaration at the interface.

```
procedure Add_To_Parts is

  insert into P (PNO, PNAME, COLOR, WEIGHT, CITY)
  values ('P02367', 'RIGHT FENDER: TOYOTA', Red, 25,
         'PITTSBURGH')
  status Operation_Map named Insert_Status
;
```

The above declaration produces the following Ada procedure declaration at the abstract interface.

```
procedure Add_To_Parts (          -- 5.8: Ada Semantics, Note
  Insert_Status : out Operation_Status);
```

**End Examples**

## 8.10 Value Expressions

The concrete syntax of value expressions differs from the concrete syntax of SQL value expressions in the following ways:

1. An operand of a value expression may be a reference to a constant, domain parameter, or enumeration literal defined either in a definitional module or in the enclosing abstract module.

2. Value expressions are strongly typed; therefore, a domain conversion operation is introduced.

```
value_expression ::=
    term | value_expression + term | value_expression - term

term ::=
    factor | term * factor | term / factor

factor ::= [+|-] primary

primary ::=
```

```
        literal  |
        constant_reference |
        domain_parameter_reference |
        column_reference |
        input_reference |
        USER |
        set_function_specification |
        domain_conversion |
        ( value_expression )

    set_function_specification ::=
        COUNT(*) | distinct_set_function | all_set_function

    distinct_set_function ::=
        {AVG | MAX | MIN | SUM | COUNT} (DISTINCT column_reference)

    all_set_function ::=
        {AVG | MAX | MIN | SUM} ([ALL] value_expression)
        see 6.5 in ISO/IEC 9075:1992

    domain_conversion ::=
        domain_reference ( value_expression )
```

Five mappings are defined on value_expressions: AdaNAME, DOMAIN, DATACLASS, LENGTH and SCALE.

The mapping AdaNAME calculates the default names of row record components when value expressions appear in select parameter lists. The range of AdaNAME is augmented by the special value NO_NAME, the value of AdaNAME for literals and non-simple names.

The mapping DOMAIN assigns a domain to each well-formed value expression. The class of domains is augmented by the special value NO_DOMAIN, the domain of literals and universal constants.

*Note*: If DOMAIN is not defined on a value expression, then the value expression is not well-formed.

The mapping DATACLASS assigns a data class to each well-formed value expression. If the expression is a literal or universal constant (or composed solely of literals and universal constants), that is, if DOMAIN(*VE*)= NO_DOMAIN, then the mapping returns the data class of the literal or universal constant (See 5.4 of this specification).

The mapping LENGTH returns the number of characters in a character string. LENGTH returns the special value NO_LENGTH on operands whose data class is not **character**.

The mapping SCALE returns the scale of a numeric expression as determined by SQL. SCALE returns the special value NO_SCALE on operands whose datatype is not numeric.

The mappings AdaNAME, DOMAIN, DATACLASS, LENGTH and SCALE are defined recursively as follows:

**Base Cases**

1. **Literals**. Let *L* be a literal. Then AdaNAME(*L*) = NO_NAME, DOMAIN(*L*) = NO_DOMAIN. DATACLASS(*L*), LENGTH(*L*) and SCALE(*L*) are defined in 5.4 of this specification.

2. **References**. Let F be an input_reference, a reference to a constant a domain_parameter_reference, or a column_reference; let G be the input parameter, constant, domain parameter or column to which F makes reference. Then

-- AdaNAME(F) = AdaID(F) if F is an identifier that is the simple name of G; otherwise, AdaNAME(F) = NO_NAME.

-- DOMAIN(F)=DOMAIN(G),

-- DATACLASS(F)=DATACLASS(G).

-- LENGTH(F)=LENGTH(G), and

-- SCALE(F)=SCALE(G),

See subclauses 8.6, 7.1.4, 7.1.3 and 7.2.1 of this specification.

3. **Keyword** USER. The following definitions apply to the value expression consisting of the keyword USER.

-- AdaNAME(USER) = NO_NAME

-- DOMAIN(USER) = NO_DOMAIN

-- DATACLASS(USER)= **character**

-- LENGTH(USER) = *implementation defined* See Syntax Rule 4 of clause 6.2 of ISO/IEC 9075:1992

-- SCALE(USER) = NO_SCALE

*Note:* These definitions imply that, when used as a select parameter, USER must appear within a domain conversion and with a named_phrase.

**Recursive Cases**

1. **Set Functions**. Let *SF* be a set function and let *VE* be a value expression.

-- AdaNAME(*SF(VE)*) = NO_NAME.

-- If *SF* is MIN or MAX, then

-- DOMAIN(*SF(VE)*) = DOMAIN(*VE*))

-- DATACLASS(*SF(VE)*) = DATACLASS(*VE*)

-- LENGTH(*SF(VE)*) = LENGTH(*VE*)

-- SCALE(*SF(VE)*) = SCALE(*VE*)

-- If *SF* is COUNT, then

-- DOMAIN(COUNT(*VE*)) = DOMAIN(COUNT(*)) = *NO_DOMAIN*

-- DATACLASS(COUNT(*VE*)) = DATACLASS(COUNT(*)) = **integer**

-- LENGTH(COUNT(*VE*))= LENGTH(COUNT(*)) = NO_LENGTH

-- SCALE(COUNT(*VE*)) = SCALE(COUNT(*)) = 0 (see 5.4 of this specification).

-- If *SF* is SUM, then

-- DOMAIN(SUM(*VE*)) = *NO_DOMAIN*

-- DATACLASS(SUM(*VE*)) = DATACLASS(*VE*) and shall be a numeric data class

-- LENGTH(SUM(*VE*)) = NO_LENGTH

-- SCALE(SUM(*VE*)) = SCALE(*VE*) (Syntax Rule 9 of 6.5 in ISO/IEC 9075:1992).

-- If *SF* is AVG, then

-- DOMAIN(AVG(*VE*)) = *NO_DOMAIN*

-- LENGTH(AVG(*VE*)) = NO_LENGTH

-- DATACLASS(*VE*) shall be numeric and DATACLASS(AVG(*VE*)) and SCALE(AVG(*VE*)) are implementation defined (Syntax Rule 9 of 6.5 in ISO/IEC 9075:1992).

2. **Domain Conversions.** Let *D* be a domain reference and *VE* a value expression. Then

-- AdaNAME(*D(VE)*) = NO_NAME.

-- DOMAIN(*D(VE)*) = *D provided*

a. DATACLASS(D) and DATACLASS(VE) are both numeric. In this case SCALE(*D(VE)*) = SCALE(*VE*), and if SCALE(D) < SCALE(VE) then a warning message shall be generated that will state, in effect, that the loss of scale implied by this conversion will not occur in the query execution. The warning message need not be generated if the value expression is in an assignment context (see 6.5 of this specification). LENGTH(*D(VE)*) = NO_LENGTH in this case.

b. DATACLASS(*D*) and DATACLASS(*VE*) are both character. In this case, LENGTH(*D(VE)*) = LENGTH(*VE*), and if LENGTH(D) < LENGTH(VE) then a warning message shall be generated that will state, in effect, that the loss of length implied by this conversion will not occur in the query execution. The warning message need not be generated if the value expression is in an assignment context (see 6.5 of this specification). SCALE(*D(VE)*)=NO_SCALE in this case.

c. DATACLASS(*D*) and DATACLASS(*VE*) are both enumeration, *provided* that

i. If DOMAIN(*VE*) ≠ NO_DOMAIN, then DOMAIN(*VE*) = D.

ii. If DOMAIN(*VE*) = NO_DOMAIN, then the value of *VE* is an enumeration literal in the domain *D*. (*Note*: Thus domain conversion may play the role played by type qualification in Ada, 4.7 in ISO 8652:1987.)

LENGTH(*D(VE)*) = NO_LENGTH and SCALE(*D(VE)*) = NO_SCALE in this case.

-- DATACLASS(*D*(*VE*)) = DATACLASS(*VE*).

*Note*: These rules imply that the equalities

-- DATACLASS(DOMAIN(*VE*)) = DATACLASS(*VE*)

-- LENGTH(DOMAIN(*VE*)) = LENGTH(*VE*)

-- SCALE (DOMAIN(*VE*)) = SCALE(*VE*)

do *not* necessarily hold.

3. **Arithmetic Operators**. Let $VE_1$, $VE_2$ be value expressions. Let

$$\begin{array}{l} \mathrm{DOMAIN}(VE_1) = D_1; \\ \mathrm{DOMAIN}(VE_2) = D_2; \\ \mathrm{DATACLASS}(VE_1) = \mathrm{T}_1; \\ \mathrm{DATACLASS}(VE_2) = \mathrm{T}_2; \\ \mathrm{SCALE}(VE_1) = \mathrm{S}_1; \\ \mathrm{SCALE}(VE_2) = \mathrm{S}_2; \end{array}$$

Then $\mathrm{T}_1$ and $\mathrm{T}_2$ shall be numeric data classes and

a. For unary operators (+, -)

-- AdaNAME($[+|-]VE_1$) = NO_NAME.

-- LENGTH($[+|-]VE_1$) = NO_LENGTH.

-- DOMAIN($[+|-]VE_1$) = $D_1$

-- DATACLASS($[+|-]VE_1$) = $\mathrm{T}_1$;

-- SCALE($[+|-]VE_1$) = $\mathrm{S}_1$;

b. Let *op* be any binary arithmetic operator. Then AdaNAME($VE_1$ *op* $VE_2$) = NO_NAME. LENGTH($VE_1$ *op* $VE_2$) = NO_LENGTH.

c. DATACLASS($VE_1$ *op* $VE_2$) = max($\mathrm{T}_1$, $\mathrm{T}_2$) where **float** > **fixed** > **integer**.

d. Recall that the DOMAIN mapping is defined for a value expression just in case that value expression is legal. The value expression $VE_1$ *op* $VE_2$ is a *legal* value expression if

-- $D_1 \neq$ NO_DOMAIN and $D_2 \neq$ NO_DOMAIN and either

-- $T_1 = T_2$ = **fixed** and op is either multiplication (*) or division (/), or

-- $D_1 = D_2$

-- or $D_1$ = NO_DOMAIN or $D_2$ = NO_DOMAIN, and

-- $T_1 = T_2$ = **integer**, or else

-- $T_1 \neq$ **integer** and then $T_2 \neq$ **integer**, or else

-- $T_1$ = **fixed** and $D_2$ = NO_DOMAIN and op is either multiplication (*) or division (/), or else

-- $T_2$ = **fixed** and $D_1$ = NO_DOMAIN and op is multiplication (*)

-- otherwise, $VE_1$ *op* $VE_2$ is not a legal value expression.

e. If $VE_1$ *op* $VE_2$ is a legal value expression, then DOMAIN($VE_1$ *op* $VE_2$) =

-- NO_DOMAIN *provided* that either

-- $D_1 = D_2$ = NO_DOMAIN

-- *or* $D_1 \neq$ NO_DOMAIN and $D_2 \neq$ NO_DOMAIN and $T_1 = T_2$ = **fixed** *and op* is either multiplication ($*$) or division (/).

-- $D_1$ *provided* that $D_1 \neq$ NO_DOMAIN

-- $D_2$ *otherwise*.

f. SCALE($VE_1$ *op* $VE_2$) is given by

-- If *op* is an additive operator (*[*+|-*]*), then the larger of $S_1$ and $S_2$.

-- If *op* is multiplication ($*$), then the sum of $S_1$ and $S_2$.

-- if *op* is division (/), then it is implementation defined. (See Syntax Rules 1, 2 of 6.12 in ISO/IEC 9075:1992.)

*Note*: The following are consequences of the definitions above.

-- AdaNAME(VE) has a value other than NO_NAME only in the case where VE is a simple identifier.

-- The product and quotient of any two **fixed** quantities with domains are always defined as **fixed** quantities with no domain, much like the Ada <universal_fixed>. However, whereas in Ada no operations other than conversion are defined for such quantities, they may be used anywhere that a literal with **fixed** data class may be used.

-- The result of a COUNT set function is treated as though it were an integer literal (see 6.5 in ISO/IEC 9075:1992).

-- The result of a SUM set function on a value expression VE is treated as though it were a literal of the data-class DATACLASS(VE)(see 6.5 in ISO/IEC 9075:1992).

-- The result of an AVG set function is treated as though it were a literal of an implementation-defined data class and scale (see 6.5 in ISO/IEC 9075:1992).

## SQL Semantics

The SQL value expression derived from a value expression *VE* is formed by removing all domain conversions and replacing all constants and domain parameters with their values and all enumeration literals with their database representations (see 7.3 of this specification). Let $SQL_{VE}$ represent the function transforming value_expressions into *SQL*_value_expressions. Let *VE* be a value_expression. $SQL_{VE}(VE)$ is given recursively as follows:

1. If *VE* contains no operators, then

a. If *VE* is a column reference or a database literal, then $SQL_{VE}(VE)$ is *VE*.

b. If *VE* is an enumeration_literal of domain D, and D assigns expression E to that enumeration literal (see rule 12 of 7.1.3 of this specification), then $SQL_{VE}(VE) = SQL_{VE}(E)$.

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

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

then $SQL_{VE}(VE) = SQL_{VE}(E)$.

d. If *VE* is a reference to a domain parameter P of domain D, and D assigns the expression E to P (see rule 6 of 7.1.3 of this specification), then $SQL_{VE}$(*VE*) = $SQL_{VE}$(E).

e. If *VE* is a reference to the input parameter, *INP*, and $PARM_{SQL}$(INP) is :C T (for C an *SQL*_Identifier and T a data type, see 8.6 of this specification), then $SQL_{VE}$(*VE*) is

:C [INDICATOR :$INDIC_{NAME}$(INP)]

where INDICATOR $INDIC_{NAME}$(INP) appears precisely when $INDIC_{SQL}$(INP) is defined. See 8.6 of this specification.

2. If *VE* is *SF*($VE_1$) where *SF* is a set function, then $SQL_{VE}$(*SF*($VE_1$)) is *SF*($SQL_{VE}$($VE_1$)).

3. If *VE* is *D*($VE_1$), where *D* is a domain name, then $SQL_{VE}$(*D*($VE_1$)) is $SQL_{VE}$($VE_1$).

4. If *VE* is +$VE_1$ (or -$VE_1$) then $SQL_{VE}$(*VE*) is +$SQL_{VE}$($VE_1$) (or -$SQL_{VE}$($VE_1$)).

5. If *VE* is $VE_1$ *op* $VE_2$ where *op* is an arithmetic operator, then $SQL_{VE}$(*VE*) is $SQL_{VE}$($VE_1$) *op* $SQL_{VE}$($VE_2$).

6. If *VE* is ($VE_1$) then $SQL_{VE}$(*VE*) is ($SQL_{VE}$($VE_1$)).

*Note*: As a consequence of these definitions, particularly item 3, a domain conversion should be considered an instruction to the processor that a given expression is well formed; it should not be considered a data conversion. Although the SAMeDL enforces a strict typing discipline, data conversions are carried out under the rules of SQL, not those of Ada. It is for this reason that warning messages are given for conversions that lose scale.

## 8.11 Search Conditions

The concrete syntax of search conditions differs from that of SQL only in that value_expression (8.10 of this specification) replaces *SQL*_value_expression in the definition of the atomic predicates (8 of ISO/IEC 9075:1992). A strict typing discipline is applied to atomic predicates.

The atomic predicates of SQL take a varying number of operands: comparison_predicate takes two, between_predicate takes three, in_predicate takes any number. So let {$OP_1$, $OP_2$, . . . , $OP_m$} be the set of operands of any atomic predicate. Each of the $OP_i$ is of the form of a value_expression. Therefore, the functions DOMAIN and DATACLASS may be applied to them (See 8.10 of this specification). For an atomic predicate to be well formed, then for any pair of distinct i and j, $1 \leq i,j \leq m$

1. If DOMAIN($OP_i$) $\neq$ NO_DOMAIN and DOMAIN($OP_j$) $\neq$ NO_DOMAIN, then DOMAIN($OP_i$) = DOMAIN($OP_j$), and

2. Exactly one of the following holds:

a. DATACLASS($OP_i$) = DATACLASS($OP_j$) = **integer**.

b. DATACLASS($OP_i$) = DATACLASS($OP_j$) = **character**.

c. both DATACLASS($OP_i$) and DATACLASS($OP_j$) are elements of the set {**fixed**, **float**}.

d. DATACLASS($OP_i$) = DATACLASS($OP_j$) = **enumeration**, and there exists some k, $1 \leq k \leq m$ such that

i. DOMAIN($OP_k$) $\neq$ NO_DOMAIN, and

ii. For all l, $1 \leq l \leq m$, either

1. DOMAIN($OP_l$) = NO_DOMAIN, and $OP_l$ is an enumeration literal of the domain DOMAIN($OP_k$), or

2. DOMAIN($OP_l$) = DOMAIN($OP_k$).

## SQL Semantics

A search condition is transformed into an *SQL*_search_condition by application of the transformation $SQL_{SC}$ that operates by executing the transformation $SQL_{VE}$, defined in 8.10 of this specification, to the value expressions appearing within the search condition and the transformation $SQL_{SQ}$, defined in 8.12 of this specification, to the subqueries in the search condition. In other words, let P, $P_1$, and $P_2$ be search conditions, VE, $VE_1$, $VE_2$, $VE_i$ be value_expressions, and SQ be a subquery. Then $SQL_{SC}$(P) is given by

1. If P is of the form: $P_1\ op\ P_2$, where $op$ is one of **and** or **or**, then $SQL_{SC}(P)$ is $SQL_{SC}(P_1)\ op\ SQL_{SC}(P_2)$. (See 8.12 of ISO/IEC 9075:1992).

2. If P is of the form: **not** $P_1$, then $SQL_{SC}(P)$ is **not** $SQL_{SC}(P_1)$. (See 8.12 of ISO/IEC 9075:1992).

3. If P is of the form: $VE_1\ op\ VE_2$, where $op$ is an SQL comparison operator, then

   $SQL_{SC}(P)=SQL_{VE}(VE_1)\ op\ SQL_{VE}(VE_2)$

   If P $=VE\ op\ SQ$, then $SQL_{SC}(P)=SQL_{VE}(VE)\ op\ SQL_{SQ}(SQ)$. (See 8.2 in ISO/IEC 9075:1992.)

4. If P is of the form: $VE$ **[not] between** $VE_1$ **and** $VE_2$ then

   $SQL_{SC}(P)=SQL_{VE}(VE)$ **[not] between** $SQL_{VE}(VE_1)$ **and** $SQL_{VE}(VE_2)$

   (see 8.2 of ISO/IEC 9075:1992).

5. If P is of the form: $VE$ **[not] in** $SQ$, then $SQL_{SC}(P)=SQL_{VE}(VE)$ **[not] in** $SQL_{SQ}(SQ)$. If P is of the form:

   $VE$ **[not] in** $(VE_1, VE_2, \ldots, VE_i, \ldots)$

   then

   $SQL_{SC}(P)=SQL_{VE}(VE)$ **[not]in** $(SQL_{VE}(VE_1), SQL_{VE}(VE_2), \ldots, SQL_{VE}(VE_i), \ldots)$

   (See 8.4 of ISO/IEC 9075:1992.).

6. If P is of the form: $VE_1$ **[not] like** $VE_2$ **escape** $c$ where *c* is a character literal of length 1, then

   $SQL_{SC}(P)=SQL_{VE}(VE_1)$ **[not] like** $SQL_{VE}(VE_2)$ **escape** $c$

   (See 8.5 in ISO/IEC 9075:1992.).

7. If P is of the form: $C$ **is [not] null** where C is an column_reference, then $SQL_{SC}(P)=C$ **is [not] null**. (See 8.6 in ISO/IEC 9075:1992.). *Note*: $SQL_{SC}$ is the identity mapping on SQL_null_predicates.

8. If P is of the form: $VE\ op\ quant\ SQ$ where $op$ is an SQL_comp_op and $quant$ is an SQL_quantifier (i.e., one of SOME, ANY or ALL), then $SQL_{SC}(P)=SQL_{VE}(VE)\ op\ quant\ SQL_{SQ}(SQ)$. (See 8.7 in ISO/IEC 9075:1992.).

9. If P is of the form: **exists** $SQ$, then $SQL_{SC}(P)=$ **exists** $SQL_{SQ}(SQ)$. (See 8.8 in ISO/IEC 9075:1992.).

## 8.12 Subqueries

The concrete syntax of a subquery differs from that of a query specification (see 8.4 of this specification) in that the select list is limited to at most one parameter. Further, that parameter, when present, takes the form of a value_expression (see 8.10 of this specification), not a select_parameter (see 8.7 of this specification), as it is not visible to the user of the abstract module.

```
subquery ::=
    ( select [distinct | all ] result_expression
        from_clause
        [where search_condition]
        [SQL_group_by_clause]
        [having search_condition] )

result_expression ::=
    value_expression | *
```

### Ada Semantics

If, within a subquery, $SQ$, result_expression takes the form of a value_expression, $VE$, then $DOMAIN(SQ)=DOMAIN(VE)$ and $DATACLASS(SQ)=DATACLASS(VE)$. $DOMAIN(SQ)$ and $DATACLASS(SQ)$ are undefined when result_expression takes the form of $*$.

*Note*: The fact that $DOMAIN(*)$ is undefined means that such a result_expression can be used only if the subquery appears within an exists_predicate.

### SQL Semantics

The *SQL*_subquery formed from a subquery, *SQ*, denoted $SQL_{SQ}(SQ)$, is produced by removing any **as** keywords, if present, from the from_clause and applying the transformation $\mathrm{SQL}_{\mathrm{SC}}$ to the search_conditions in the **where** and **having** clauses, if present.

## 8.13 Status Clauses

A status clause serves to attach a status map to a procedure and optionally rename the status parameter.

```
status_clause ::=
    status status_reference [named_phrase]
```

### Ada Semantics

If a procedure P has a status_clause of the form

```
status M [named Id_1]
```

and the definition of *M* was given by

```
enumeration T is (L1, ..., Ln);

status M
    [named Id_2]
    uses T
    is ( ..., n => L, ... );
```

(see 7.1.8 of this specification),

then:

1. The procedure $P_{Ada}$ (see 8.2 and 8.5 of this specification) shall have a status parameter of type *T*.

2. The name of the status parameter of $P_{Ada}$ is determined by:

   a. If *Id_1* is present in the status_clause, then the name of the status parameter shall be *Id_1*.

   b. If rule (a) does not apply, then if *Id_2* is present in the definition of the status map *M*, the name of the status parameter shall be *Id_2* (see 7.1.8 of this specification).

   c. If neither rule (a) nor rule (b) applies, then the name of the status parameter shall be *Status*.

# 9 Conformance

## 9.1 Introduction

This International Standard specifies conforming SAMeDL language and conforming SAMeDL implementations.

Conforming SAMeDL language shall abide by the BNF Format and associated discussion of this specification.

A conforming SAMeDL implementation shall process conforming SAMeDL language according to the Ada Semantics, SQL Semantics and Interface Semantics of this specification.

## 9.2 Claims of Conformance

### 9.2.1 Introduction

Claims of conformance to this International Standard shall be to one or more of the following binding styles:

a) conformance via mapping;
b) conformance via effects.

### 9.2.2 Conformance via mapping.

An implementation claiming conformance via mapping shall, for each abstract module, generate an Ada library unit package satisfying the Ada semantics of clause 8.1 of this specification. In addition, an implementation claiming conformance via mapping shall specify one of the following binding substyles:

i) conformance via SQL module;
ii) conformance via embedded SQL Syntax.

#### 9.2.2.1 Conformance via SQL module.

An implementation claiming conformance via SQL module shall, for each abstract module, generate an SQL module satisfying the SQL semantics of clause 8.1 of this specification.

#### 9.2.2.2 Conformance via embedded SQL Syntax.

An implementation claiming conformance via embedded SQL Syntax shall, for each abstract module, generate one or more embedded SQL host programs or program fragments, which program(s) or program fragments(s) shall conform to ISO/IEC 9075:1992. Such an implementation shall state which of the host programming languages listed in clause 19 of ISO/IEC 9075:1992 it generates.

### 9.2.3 Conformance via effects.

Although the processing of abstract modules is defined in terms of the derivation of a program conforming to the Ada programming language standard ISO 8652:1987 and the SQL Database Language ISO/IEC 9075:1992, implementations of SAMeDL claiming conformance via effects are not constrained to follow that method, provided the same effect is achieved.

### 9.2.4 Multiple claims of conformance.

An implementation claiming conformance in more than one of the styles of clause 9.2.1 may provide a user option as to which style is to be used.

## 9.3 Extensions

Conforming SAMeDL implementations shall not process SAMeDL language that is not conforming. Implementation defined facilities may be included into conforming SAMeDL via the extensions facility provided by the SAMeDL (see 4.2.9 of this specification). 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 contain implementation defined functionality.

Implementation defined functionality introduced by extensions may imply an implementation defined extension of the list of reserved words and thereby may prevent proper processing of some texts that otherwise conform to the requirements of this specification.

# Annex A: SAMeDL_Standard

The predefined SAMeDL definitional module SAMeDL_Standard provides a common location for declarations that are standard for all implementations of the SAMeDL. This definitional module includes the SAMeDL declarations for

-- the exceptions SQL_Database_Error and Null_Value_Error (see 6.6 and 7.3 of this specification);

-- the SAMeDL standard base domains;

-- integer constants whose values are the SQLCODE values predefined in ISO/IEC 9075:1992;

-- the domains SQLSTATE_Domain and SQLSTATE_Class_Domain and constants of those domains, mirroring the constants defined in ISO/IEC 9075:1992;

-- the pre-defined Status Map Standard_Map.

```
definition module SAMeDL_Standard is

    exception SQL_Database_Error;
    exception Null_Value_Error;

    -- SQL_Int is based on the Ada type SQL_Standard.Int
    base domain SQL_Int
        (first : integer;
         last  : integer)
    is
        domain pattern is
            'type [self]_Not_Null is new SQL_Int_Not_Null'
               '{ range [first] .. [last]};'
            'type [self]_Type is new SQL_Int;'
            'package [self]_Ops is new SQL_Int_Ops('
               '[self]_Type, [self]_Not_Null);'
        end pattern;

        derived domain pattern is
            'type [self]_Not_Null is new [parent]_Not_Null'
               '{ range [first] .. [last]};'
               'type [self]_Type is new [parent]_Type;'
            'package [self]_Ops is new SQL_Int_Ops('
               '[self]_Type, [self]_Not_Null);'
        end pattern;

        subdomain pattern is
            'subtype [self]_Not_Null is [parent]_Not_Null'
               '{ range [first] .. [last]};'
            'type [self]_Type is new [parent]_Type;'
            'package [self]_Ops is new SQL_Int_Ops('
               '[self]_Type, [self]_Not_Null);'
        end pattern;
```

```
    for not null type name use '[self]_Not_Null';
    for null type name use '[self]_Type';
    for data class use integer;
    for dbms type use integer;
    for conversion from dbms to not null use type mark;
    for conversion from not null to null use function
       '[self]_Ops.With_Null';
    for conversion from null to not null use function
       '[self]_Ops.Without_Null';
    for conversion from not null to dbms use type mark;

end SQL_Int;
```

```
-- SQL_Smallint is based on the Ada type SQL_Standard.Smallint
base domain SQL_Smallint
    (first : integer;
     last  : integer)
is

    domain pattern is
       'type [self]_Not_Null is new SQL_Smallint_Not_Null'
          '{ range [first] .. [last]};'
       'type [self]_Type is new SQL_Smallint;'
       'package [self]_Ops is new SQL_Smallint_Ops('
          '[self]_Type, [self]_Not_Null);'
    end pattern;

    derived domain pattern is
       'type [self]_Not_Null is new [parent]_Not_Null'
          '{ range [first] .. [last]};'
       'type [self]_Type is new [parent]_Type;'
       'package [self]_Ops is new SQL_Smallint_Ops('
          '[self]_Type, [self]_Not_Null);'
    end pattern;

    subdomain pattern is
       'subtype [self]_Not_Null is [parent]_Not_Null'
          '{ range [first] .. [last]};'
       'type [self]_Type is new [parent]_Type;'
       'package [self]_Ops is new SQL_Smallint_Ops('
          '[self]_Type, [self]_Not_Null);'
    end pattern;

    for not null type name use '[self]_Not_Null';
    for null type name use '[self]_Type';
    for data class use integer;
    for dbms type use smallint;
    for conversion from dbms to not null use type mark;
    for conversion from not null to null use function
       '[self]_Ops.With_Null';
    for conversion from null to not null use function
       '[self]_Ops.Without_Null';
    for conversion from not null to dbms use type mark;

end SQL_Smallint;
```

```
-- SQL_Real is based on the Ada type SQL_Standard.Real
base domain SQL_Real
    (first : float;
     last  : float)
is

    domain pattern is
       'type [self]_Not_Null is new SQL_Real_Not_Null'
          '{ range [first] .. [last]};'
       'type [self]_Type is new SQL_Real;'
       'package [self]_Ops is new SQL_Real_Ops('
          '[self]_Type, [self]_Not_Null);'
    end pattern;

    derived domain pattern is
       'type [self]_Not_Null is new [parent]_Not_Null'
          '{ range [first] .. [last]};'
       'type [self]_Type is new [parent]_Type;'
       'package [self]_Ops is new SQL_Real_Ops('
          '[self]_Type, [self]_Not_Null);'
    end pattern;

    subdomain pattern is
       'subtype [self]_Not_Null is [parent]_Not_Null'
          '{ range [first] .. [last]};'
       'type [self]_Type is new [parent]_Type;'
       'package [self]_Ops is new SQL_Real_Ops('
          '[self]_Type, [self]_Not_Null);'
    end pattern;

    for not null type name use '[self]_Not_Null';
    for null type name use '[self]_Type';
    for data class use float;
    for dbms type use real;
    for conversion from dbms to not null use type mark;
    for conversion from not null to null use function
       '[self]_Ops.With_Null';
    for conversion from null to not null use function
       '[self]_Ops.Without_Null';
    for conversion from not null to dbms use type mark;

end SQL_Real;
```

```
    -- SQL_Double_Precision is based on the Ada type
    -- SQL_Standard.Double_Precision
    base domain SQL_Double_Precision
        (first : float;
         last  : float)
is

        domain pattern is
           'type [self]_Not_Null is new SQL_Double_Precision_Not_Null'
              '{ range [first] .. [last]};'
           'type [self]_Type is new SQL_Double_Precision;'
           'package [self]_Ops is new SQL_Double_Precision_Ops('
              '[self]_Type, [self]_Not_Null);'
        end pattern;

        derived domain pattern is
           'type [self]_Not_Null is new [parent]_Not_Null'
              '{ range [first] .. [last]};'
           'type [self]_Type is new [parent]_Type;'
           'package [self]_Ops is new SQL_Double_Precision_Ops('
              '[self]_Type, [self]_Not_Null);'
        end pattern;

        subdomain pattern is
           'subtype [self]_Not_Null is [parent]_Not_Null'
              '{ range [first] .. [last]};'
           'type [self]_Type is new [parent]_Type;'
           'package [self]_Ops is new SQL_Double_Precision_Ops('
              '[self]_Type, [self]_Not_Null);'
        end pattern;

        for not null type name use '[self]_Not_Null';
        for null type name use '[self]_Type';
        for data class use float;
        for dbms type use double precision;
        for conversion from dbms to not null use type mark;
        for conversion from not null to null use function
           '[self]_Ops.With_Null';
        for conversion from null to not null use function
           '[self]_Ops.Without_Null';
        for conversion from not null to dbms use type mark;

    end SQL_Double_Precision;
```

```
-- SQL_Char is based on the Ada type SQL_Standard.Char
base domain SQL_Char
is

    domain pattern is
       'type [self]NN_Base is new SQL_Char_Not_Null;'
       'subtype [self]_Not_Null is [self]NN_Base (1 .. [length]);'
       'type [self]_Base is new SQL_Char;'
       'subtype [self]_Type is [self]_Base ('
          '[self]_Not_Null''length);'
       'package [self]_Ops is new SQL_Char_Ops('
          '[self]_Base, [self]NN_Base);'
    end pattern;

    derived domain pattern is
       'type [self]NN_Base is new [parent]NN_Base;'
       'subtype [self]_Not_Null is [self]NN_Base (1 .. [length]);'
       'type [self]_Base is new [parent]_Base;'
       'subtype [self]_Type is [self]_Base ('
          '[self]_Not_Null''length);'
       'package [self]_Ops is new SQL_Char_Ops ('
          '[self]_Base, [self]NN_Base);'
    end pattern;

    subdomain pattern is
       'subtype [self]NN_Base is [parent]NN_Base;'
       'subtype [self]_Not_Null is [parent]NN_Base (1 .. [length]);'
       'subtype [self]_Base is new [parent]_Base;'
       'subtype [self]_Type is [self]_Base ('
          '[self]_Not_Null''length);'
       'package [self]_Ops is new SQL_Char_Ops ('
          '[self]_Base, [self]NN_Base);'
    end pattern;

    for not null type name use '[self]_Not_Null';
    for null type name use '[self]_Type';
    for data class use character;
    for dbms type use character '([length])';
    for conversion from dbms to not null use type mark;
    for conversion from not null to null use function
       '[self]_Ops.With_Null';
    for conversion from null to not null use function
       '[self]_Ops.Without_Null';
    for conversion from not null to dbms use type mark;

end SQL_Char;
```

```
-- SQL_Enumeration_As_Int is based on the Ada type
-- SQL_Standard.Int
base domain SQL_Enumeration_As_Int
    (map := pos)
is

    domain pattern is
       'type [self]_Not_Null is new [enumeration];'
       'package [self]_Pkg is new SQL_Enumeration_Pkg('
          '[enumeration]);'
       'type [self]_Type is new [self]_Pkg.SQL_Enumeration;'
    end pattern;

    derived domain pattern is
       'type [self]_Not_Null is new [parent]_Not_Null;'
       'type [self]_Type is new [parent]_Type;'
    end pattern;

    subdomain pattern is
       'subtype [self]_Not_Null is [parent]_Not_Null;'
       'subtype [self]_Type is [parent]_Type;'
    end pattern;

    for not null type name use '[self]_Not_Null';
    for null type name use '[self]_Type';
    for data class use enumeration;
    for dbms type use integer;
    for conversion from not null to null use function
       '[self]_Pkg.With_Null';
    for conversion from null to not null use function
       '[self]_Pkg.Without_Null';

end SQL_Enumeration_As_Int;
```

```
-- SQL_Enumeration_As_Char is based on the Ada type
-- SQL_Standard.Char
base domain SQL_Enumeration_As_Char
    (map := image)
is

    domain pattern is
       'type [self]_Not_Null is new [enumeration];'
       'package [self]_Pkg is new SQL_Enumeration_Pkg('
          '[enumeration]);'
       'type [self]_Type is new [self]_Pkg.SQL_Enumeration;'
    end pattern;

    derived domain pattern is
       'type [self]_Not_Null is new [parent]_Not_Null;'
       'type [self]_Type is new [parent]_Type;'
    end pattern;

    subdomain pattern is
       'subtype [self]_Not_Null is [parent]_Not_Null;'
       'subtype [self]_Type is [parent]_Type;'
    end pattern;

    for not null type name use '[self]_Not_Null';
    for null type name use '[self]_Type';
    for data class use enumeration;
    for dbms type use character '([length])';
    for conversion from not null to null use function
       '[self]_Pkg.With_Null';
    for conversion from null to not null use function
       '[self]_Pkg.Without_Null';

end SQL_Enumeration_As_Char;
```

```
-- SQLCODE Standard values
-- SQLSTATE domains and constants

    -- SQLCODE for successful execution of a statement
constant Success is 0;

    --  SQLCODE for data not found
constant Not_Found is 100;

-- Domain declarations for SQLSTATE values

domain SQLSTATE_Domain is new SQL_Char not null
    (Length => 5);
domain SQLSTATE_Class_Domain  is new SQL_Char not null
    (Length => 2);

-- These SQLSTATE values and their names
-- are taken from the definition of the Ada package SQL_Standard,
-- defined in 12.4 of ISO/IEC 9075:1992.
-- Any discrepency between this list and the list in the citation
-- is resolved in favor of the later, which is definitive.

constant Ambiguous_Cursor_Name : SQLSTATE_Class_Domain is '3C';
constant Ambiguous_Cursor_Name_No_Subclass :
    SQLSTATE_Domain is '3C000';

constant Cardinality_Violation : SQLSTATE_Class_Domain is '21';
constant Cardinality_Violation_No_Subclass :
    SQLSTATE_Domain is '21000';

constant Connection_Exception : SQLSTATE_Class_Domain is '08';
constant Connection_Exception_No_Subclass :
    SQLSTATE_Domain is '08000';
constant Connection_Exception_Connection_Does_Not_Exist :
    SQLSTATE_Domain is '08003';
constant Connection_Exception_Connection_Failure :
    SQLSTATE_Domain is '08006';
constant Connection_Exception_Connection_Name_In_Use :
    SQLSTATE_Domain is '08002';
constant Connection_Exception_SQLClient_Unable_To_Establish_SQLConnection :
    SQLSTATE_Domain is '08001';
constant Connection_Exception_SQLServer_Rejected_Establishment_Of_SQLConnection :
    SQLSTATE_Domain is '08004';
constant Connection_Exception_Transaction_Resolution_Unknown :
    SQLSTATE_Domain is '08007';

constant Data_Exception: SQLSTATE_Class_Domain is '22';
constant Data_Exception_No_Subclass :
    SQLSTATE_Domain is '22000';
constant Data_Exception_Character_Not_in_Repertoire :
    SQLSTATE_Domain is '22008';
constant Data_Exception_DateTime_Field_Overflow :
    SQLSTATE_Domain is '22008';
constant Data_Exception_Division_By_Zero :
    SQLSTATE_Domain is '22012';
constant Data_Exception_Error_In_Assignment :
    SQLSTATE_Domain is '22005';
constant Data_Exception_Indicator_Overflow :
    SQLSTATE_Domain is '22022';
constant Data_Exception_Interval_Field_Overflow :
    SQLSTATE_Domain is '22015';
```

```
constant Data_Exception_Invalid_Character_Value_For_Cast :
    SQLSTATE_Domain is '22018';
constant Data_Exception_Invalid_DateTime_Format :
    SQLSTATE_Domain is '22007';
constant Data_Exception_Invalid_Escape_Character :
    SQLSTATE_Domain is '22019';
constant Data_Exception_Invalid_Escape_Sequence :
    SQLSTATE_Domain is '22025';
constant Data_Exception_Invalid_Parameter_Value :
    SQLSTATE_Domain is '22023';
constant Data_Exception_Invalid_Time_Zone_Displacement_Value :
    SQLSTATE_Domain is '22009';
constant Data_Exception_Null_Value_No_Indicator_Parameter :
    SQLSTATE_Domain is '22002';
constant Data_Exception_Numeric_Value_Out_of_Range :
    SQLSTATE_Domain is '22003';
constant Data_Exception_String_Data_Length_Mismatch :
    SQLSTATE_Domain is '22026';
constant Data_Exception_String_Data_Right_Truncation :
    SQLSTATE_Domain is '22001';
constant Data_Exception_Substring_Error :
    SQLSTATE_Domain is '22011';
constant Data_Exception_Trim_Error :
    SQLSTATE_Domain is '22027';
constant Data_Exception_Unterminated_C_String :
    SQLSTATE_Domain is '22024';

constant Dependent_Privilege_Descriptors_Still_Exist:
          SQLSTATE_Class_Domain is '2B';
constant Dependent_Privilege_Descriptors_Still_Exist_No_Subclass :
    SQLSTATE_Domain is '2B000';

constant Dynamic_SQL_Error : SQLSTATE_Class_Domain is '07';
constant Dynamic_SQL_Error_No_Subclass :
    SQLSTATE_Domain is '07000';
constant Dynamic_SQL_Error_Cursor_Specification_Cannot_Be_Executed :
    SQLSTATE_Domain is '07003';
constant Dynamic_SQL_Error_Invalid_Descriptor_Count :
    SQLSTATE_Domain is '07008';
constant Dynamic_SQL_Error_Invalid_Descriptor_Index :
    SQLSTATE_Domain is '07009';
constant Dynamic_SQL_Error_Prepared_Statement_Not_A_Cursor_Specification :
    SQLSTATE_Domain is '07005';
constant Dynamic_SQL_Error_Restricted_Data_Type_Attribute_Violation :
    SQLSTATE_Domain is '07006';
constant Dynamic_SQL_Error_Using_Clause_Does_Not_Match_Dynamic_Parameter_Spec :
    SQLSTATE_Domain is '07001';
constant Dynamic_SQL_Error_Using_Clause_Does_Not_Match_Target_Spec :
    SQLSTATE_Domain is '07002';
constant Dynamic_SQL_Error_Using_Clause_Required_For_Dynamic_Parameters :
    SQLSTATE_Domain is '07004';
constant Dynamic_SQL_Error_Using_Clause_Required_For_Result_Fields :
    SQLSTATE_Domain is '07007';

constant Feature_Not_Supported : SQLSTATE_Class_Domain is '0A';
constant Feature_Not_Supported_No_Subclass :
    SQLSTATE_Domain is '0A000';
constant Feature_Not_Supported_Mutliple_Environment_Transactions :
    SQLSTATE_Domain is '0A001';

constant Integriy_Constraint_Violation : SQLSTATE_Class_Domain is '23';
```

```
constant Integriy_Constraint_Violation_No_Subclass :
    SQLSTATE_Domain is '23000';

constant Invalid_Authorization_Specification : SQLSTATE_Class_Domain is '28';
constant Invalid_Authorization_Specification_No_Subclass :
    SQLSTATE_Domain is '28000';

constant Invalid_Catalog_Name : SQLSTATE_Class_Domain is '3D';
constant Invalid_Catalog_Name_No_Subclass :
    SQLSTATE_Domain is '3D000';

constant Invalid_Character_Set_Name : SQLSTATE_Class_Domain is '2C';
constant Invalid_Character_Set_Name_No_Subclass :
    SQLSTATE_Domain is '2C000';

constant Invalid_Condition_Number : SQLSTATE_Class_Domain is '35';
constant Invalid_Condition_Number_No_Subclass :
    SQLSTATE_Domain is '35000';

constant Invalid_Connection_Name : SQLSTATE_Class_Domain is '2E';
constant Invalid_Connection_Name_No_Subclass :
    SQLSTATE_Domain is '2E000';

constant Invalid_Cursor_Name : SQLSTATE_Class_Domain is '34';
constant Invalid_Cursor_Name_No_Subclass :
    SQLSTATE_Domain is '34000';

constant Invalid_Cursor_State : SQLSTATE_Class_Domain is '24';
constant Invalid_Cursor_State_No_Subclass :
    SQLSTATE_Domain is '24000';

constant Invalid_Schema_Name : SQLSTATE_Class_Domain is '3F';
constant Invalid_Schema_Name_No_Subclass :
    SQLSTATE_Domain is '3F000';

constant Invalid_SQL_Descriptor_Name : SQLSTATE_Class_Domain is '33';
constant Invalid_SQL_Descriptor_Name_No_Subclass :
    SQLSTATE_Domain is '33000';

constant Invalid_SQL_Statement_Name : SQLSTATE_Class_Domain is '26';
constant Invalid_SQL_Statement_Name_No_Subclass :
    SQLSTATE_Domain is '26000';

constant Invalid_Transaction_State : SQLSTATE_Class_Domain is '25';
constant Invalid_Transaction_State_No_Subclass :
    SQLSTATE_Domain is '25000';

constant Invalid_Transaction_Termination : SQLSTATE_Class_Domain is '2D';
constant Invalid_Transaction_Termination_No_Subclass :
    SQLSTATE_Domain is '2D000';

constant No_Data : SQLSTATE_Class_Domain is '02';
constant No_Data_No_Subclass :
    SQLSTATE_Domain is '02000';

constant Remote_Database_Access : SQLSTATE_Class_Domain is 'HZ';
constant Remote_Database_Access_No_Subclass :
    SQLSTATE_Domain is 'HZ000';

constant Successful_Completion : SQLSTATE_Class_Domain is '00';
constant Successful_Completion_No_Subclass :
```

```
    SQLSTATE_Domain is '00000';

constant Syntax_Error_Or_Access_Rule_Violation : SQLSTATE_Class_Domain is '42';
constant Syntax_Error_Or_Access_Rule_Violation_No_Subclass :
    SQLSTATE_Domain is '42000';

constant Syntax_Error_Or_Access_Rule_Violation_In_Direct_Statement:
          SQLSTATE_Class_Domain is '2A';
constant Syntax_Error_Or_Access_Rule_Violation_In_Direct_Statement_No_Subclass :
    SQLSTATE_Domain is '2A000';

constant Syntax_Error_Or_Access_Rule_Violation_In_Dynamic_Statement:
          SQLSTATE_Class_Domain is '37';
constant Syntax_Error_Or_Access_Rule_Violation_In_Dynamic_Statement_No_Subclass :
    SQLSTATE_Domain is '37000';

constant Transaction_Rollback : SQLSTATE_Class_Domain is '40';
constant Transaction_Rollback_No_Subclass :
    SQLSTATE_Domain is '40000';
constant Transaction_Rollback_Integrity_Constraint_Violation :
    SQLSTATE_Domain is '40002';
constant Transaction_Rollback_Serialization_Failure :
    SQLSTATE_Domain is '40001';
constant Transaction_Rollback_Statement_Completion_Unknown :
    SQLSTATE_Domain is '40003';

constant Triggered_Data_Change_Violation : SQLSTATE_Class_Domain is '27';
constant Triggered_Data_Change_Violation_No_Subclass :
    SQLSTATE_Domain is '27000';

constant Warning : SQLSTATE_Class_Domain is '01';
constant Warning_No_Subclass :
    SQLSTATE_Domain is '01000';
constant Warning_Cursor_Operation_Conflict :
    SQLSTATE_Domain is '01001';
constant Warning_Disconnect_Error :
    SQLSTATE_Domain is '01002';
constant Warning_Implicit_Zero_Bit_Padding :
    SQLSTATE_Domain is '01008';
constant Warning_Insufficient_Item_Descriptor_Areas :
    SQLSTATE_Domain is '01005';
constant Warning_Null_Value_Eliminated_in_Set_Function :
    SQLSTATE_Domain is '01003';
constant Warning_Privilege_Not_Granted :
    SQLSTATE_Domain is '01007';
constant Warning_Privilege_Not_Revoked :
    SQLSTATE_Domain is '01006';
constant Warning_Query_Expression_Too_Long_For_Information_Schema :
    SQLSTATE_Domain is '0100A';
constant Warning_Search_Condition_Too_Long_For_Information_Schema :
    SQLSTATE_Domain is '01009';
constant Warning_String_Data_Right_Truncation_Warning :
    SQLSTATE_Domain is '01004';

constant With_Check_Option_Violation : SQLSTATE_Class_Domain is '44';
constant With_Check_Option_Violation_No_Subclass :
    SQLSTATE_Domain is '44000';
```

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

end SAMeDL_Standard;
```

# Annex B: SAMeDL_System

The predefined SAMeDL definitional module SAMeDL_System provides a common location for the declaration of implementation-defined constants that are specific to a particular DBMS/Ada compiler platform.

```
with SAMeDL_Standard; use SAMeDL_Standard;
definition module SAMeDL_System is

    -- Smallest (most negative) value of any integer type
    constant Min_Int is implementation defined;
    -- Largest (most positive) value of any integer type
    constant Max_Int is implementation defined;

    -- Smallest value of any SQL_Int type
    constant Min_SQL_Int is implementation defined;
    -- Largest value of any SQL_Int type
    constant Max_SQL_Int is implementation defined;

    -- Smallest value of any SQL_Smallint type
    constant Min_SQL_Smallint is implementation defined;
    -- Largest value of any SQL_Smallint type
    constant Max_SQL_Smallint is implementation defined;

    -- Largest value allowed for the number of significant decimal
    -- digits in any floating point constraint
    constant Max_Digits is implementation defined;

    -- Largest value allowed for the number of significant decimal
    -- digits in an SQL_Real floating point constraint
    constant SQL_Real_Digits is implementation defined;

    -- Largest value allowed for the number of significant decimal
    -- digits in an SQL_Double_Precision floating point constraint
    constant SQL_Double_Precision_Digits is implementation defined;

    -- Largest value allowed for the number of characters in a
    -- character string constraint
    constant Max_SQL_Char_Length is implementation defined;

    -- implementation defined SQLSTATE and SQLCODE values
    -- should be declared as constants here

    end SAMeDL_System;
```

# Annex C: Standard Support Operations and Specifications

The following two sections discuss the SAME standard support packages. The first section describes how they support the standard base domains, and the second section lists their Ada package specifications.

## C.1 Standard Base Domain Operations

The SAME standard support packages encapsulate the Ada type definitions of the standard base domains, as well as the operations that provide the data semantics for domains declared using these base domains. This section describes the nature of the support packages, namely the Ada data types and the operations on objects of these types.

The SQL standard package SQL_Standard (see C.2.1 in this International Standard and 12.3.8.a.iii of ISO/IEC 9075:1992; contains the type definitions for a DBMS platform that define the Ada representations of the concrete SQL data types. A standard base domain exists in the SAMeDL for each type in SQL_Standard (except for SQLCode_Type), and these base domains are each supported by one of the SAME standard support packages. In addition to the above base domains, two standard base domains exist that provide data semantics for Ada enumeration types.

Each support package defines a not null-bearing and a null-bearing type for the base domain. The not null-bearing type is a visible Ada type derived from the corresponding type in SQL_Standard with no added constraints. This type provides the Ada application programmer with Ada data semantics for data in the database. The null-bearing type is an Ada limited private type used to support data semantics of the SQL null value. In particular, the null-bearing type may contain the null value; the not null-bearing type may not.

Domains are derived from base domains by the declaration of two Ada data types, derived from the types in the support packages, and the instantiation of the generic operations package with these types. The type derivations and the package instantiation provide the domain with the complete set of operations that define the data semantics for that domain. These operations are described below, grouped by data class.

### C.1.1 All Domains

All domains derived from the standard base domains make an *Assign* procedure available to the application because the type that supports the SQL data semantics is an Ada limited private type. For the numeric domains, this procedure enforces the range constraints that are specified for the domain when it is declared. The Ada *Constraint_Error* exception is raised by these procedures if the value to be assigned falls outside of the specified range.

A parameterless function named *Null_SQL_<type>* is available for all domains as well. This function returns an object of the null-bearing type of the appropriate domain whose value is the SQL null value.

Every domain has a set of conversion functions available for converting between the not null-bearing type and the null-bearing type. The function *With_Null* converts an object of the not null-bearing type to an object of the null-bearing type. The function *Without_Null* converts an object of the null-bearing type to an object of the not null-bearing type. *Without_Null* will raise the *Null_Value_Error* exception if the value of the object that it is converting is the SQL null value, since an object of the not null-bearing type can never be null.

Two testing functions are available for each domain as well. The boolean functions *Is_Null* and *Not_Null* test objects of the null-bearing type, returning the appropriate boolean value indicating whether or not an object contains the SQL null value.

Additionally, all domains provide two sets of comparison operators that operate on objects of the null-bearing type. The first set of operators returns boolean values, and the second set of operators returns objects of the type Boolean_With_Unknown, defined in the support package SQL_Boolean_Pkg (see Section C.2.2), which implements three-valued logic (see [1]). The boolean comparison operators are =, /=, <, >, <=, and >=, and return the value *False* if either of the objects contains the SQL null value.[1] Otherwise, these operators perform the comparison, and return the appropriate boolean result. The *Boolean_With_Unknown* comparison operators are *Equals* and *Not_Equals*, <, >, <=, and >=, and return the value *Unknown* if either of the objects contains the SQL null value. Otherwise, these operators perform the comparison, and return the *Boolean_With_Unknown* values *True* or *False*.

### C.1.2 Numeric Domains

In addition to the operations mentioned above, all numeric domains provide unary and binary arithmetic operations for the null-bearing type of the domain. The subprograms that implement these operations provide the data semantics of the SQL null value with respect to these arithmetic operations. Specifically, any arithmetic operation applied to a null value results in the null value. Otherwise, the operation is defined to be the same as the Ada operation. The unary operations that are provided are +, -, and *Abs*. The binary operations include +, -, *, and /. Finally, all numeric domains provide the exponentiation operation (**).

### C.1.3 Int and Smallint Domains

*Int* and *Smallint* domains provide the application programmer with the Ada functions *Mod* and *Rem* that operate on objects of the null-bearing type. Again, the subprograms that implement these operations provide the data semantics of the SQL null value with respect to these arithmetic operations. As with the other arithmetic operations, *Mod* and *Rem* return the null value when applied to an object containing the null value. Otherwise, they are defined to be the same as the Ada operation.

These domains also make *Image* and *Value* functions available to the application programmer. Both of these functions are overloaded, meaning that there are *Image* and *Value* functions that operate on objects of both the not null-bearing and the null-bearing types of the domain. The *Image* function converts an object of an *Int* or *Smallint* domain to a character representation of the integer value. The *Value* function converts a character representation of an integer value to an object of an *Int* or *Smallint* domain. These functions perform the same operation as the Ada attribute functions of the same name, except that the character set of the character inputs and outputs is that of the underlying SQL_Standard.Char character set. If the *Image* and *Value* functions are applied to objects of the null-bearing type containing the null value, a null character object and a null integer object are returned respectively.

[1]Note: These semantics have a peculiar side effect, namely that, for objects $O_1$ and $O_2$, the boolean expression ($O_1 < O_2$) or else ($O_1 >= O_2$) is not a tautology.

### C.1.4 Character Domains

In addition to the operations provided by all domains, character domains provide the application programmer with some string manipulation and string conversion operations.

Character domains provide two string manipulation functions that operate on objects of the null-bearing type. The first one is the catenation function (&). If either of the input character objects contains the null value, then the object returned contains the null value. Otherwise this operation is the same as the Ada catenation operation. The other function is the *Substring* function, which is patterned after the substring function in ISO/IEC 9075:1992. This function returns the portion of the input character object specified by the *Start* and *Length* index inputs. An Ada *Constraint_Error* is raised if the substring specification is not contained entirely within the input string.

The remaining operations provided by the character domains are conversion functions. A *To_String* and a *To_Unpadded_String* function exist for both the not null-bearing and the null-bearing types of the domain. The *To_String* function converts its input, which exists as an object whose value is comprised of characters from the underlying character set of the platform, to an object of the Ada predefined type *Standard.String*. If conversion of a null-bearing object containing the null value is attempted, the *Null_Value_Error* exception is raised. The *To_Unpadded_String* functions are identical in every way to the *To_String* functions, except that trailing blanks are stripped from the value.

The *Without_Null_Unpadded* function is identical to the *Without_Null* function, described in section C.1.1 above, except that trailing blanks are stripped from the value.

Two functions exist that convert objects of the Ada predefined type *Standard.String* to objects of the not null-bearing and null-bearing types of the domain. The *To_SQL_Char_Not_Null* function converts an object of type *Standard.String* to the not null-bearing type of the domain. The *To_SQL_Char* function converts an object of type *Standard.String* to an object of the null-bearing type.

Finally, character domains provide the function *Unpadded_Length*, which returns the length of the character string representation without trailing blanks. This function operates on objects of the null-bearing type, and raises the *Null_Value_Error* exception if the input object contains the null value.

### C.1.5 Enumeration Domains

Enumeration domains provide functions for the null-bearing type that are normally available as Ada attribute functions for the not null-bearing type. The *Image* and *Value* functions have the same semantics as described for *Int* and *Smallint* domains in Section C.1.3 above, except that they operate on enumeration values rather than integers.

The *Pred* and *Succ* functions operate on objects of the null-bearing type, and return the previous and next enumeration literals of the underlying enumeration type, respectively. If these functions are applied to objects containing the null value, an object containing the null value is returned.

The last two functions are the *Pos* and *Val* functions. These functions also operate on objects of the null-bearing type. *Pos* returns a value of the Ada predefined type *Standard.Integer* representing the position (relative to zero) of the enumeration literal that is the value of the input object. If the input object contains the null value, then the

*Null_Value_Error* exception is raised. The *Val* function accepts a value of the predefined type *Standard.Integer* and returns the enumeration literal whose position in the underlying enumeration type is specified by that value. If the input integer value falls outside the range of available enumeration literals, the Ada *Constraint_Error* is raised.

### C.1.6 Boolean Functions

The SAME standard support package SQL_Boolean_Pkg defines a number of boolean functions, namely *not*, *and*, *or*, and *xor*, which implement three-valued logic as defined in ISO/IEC 9075:1992. All of these functions operate on two input parameters of the type *Boolean_With_Unknown*, and return a value of that type.

This support package also provides a conversion function, which converts the input of the type *Boolean_With_Unknown* to a value of the Ada predefined type boolean. If the input object has the value *Unknown*, then the *Null_Value_Error* exception is raised.

Finally, the package provides three testing functions that return boolean values. These functions, *Is_True*, *Is_False*, and *Is_Unknown*, return the value true if the input passes the test; otherwise the functions return the value false.

### C.1.7 Operations Available to the Application

| | **Operand Type** | | | **Exceptions** |
|---|---|---|---|---|
| | **Left** | **Right** | **Result** | |
| **All Domains** | | | | |
| Null_SQL_<type> | | | _Type | |
| With_Null | | _Not_Null | _Type | |
| Without_Null | | _Type[1] | _Not_Null[2] | Null_Value_Error |
| Is_Null, Not_Null | | _Type | Boolean | |
| Assign[3] | _Type | _Type | | Constraint_Error |
| Equals, Not_Equals | _Type | _Type | B_W_U[4] | |
| <, >, <=, >= | _Type | _Type | B_W_U | |
| =, /=, >, <, >=, <= | _Type | _Type | Boolean | |
| **Numeric Domains** | | | | |
| unary +, -, Abs | | _Type | _Type | |
| +, -, /, * | _Type | _Type | _Type | |
| ** | _Type | Integer | _Type | |
| **Int and Smallint Domains** | | | | |
| Mod, Rem | _Type | _Type | _Type | |
| Image | _Type | | SQL_Char | |
| Image | _Not_Null | | SQL_ChrNN[5] | |
| Value | SQL_Char | | _Type | |
| Value | SQL_ChrNN | | _Not_Null | |
| **Character Domains** | | | | |
| Without_Null_Unpadded | | _Type | _Not_Null | Null_Value_Error |
| To_String | | _Not_Null | String | |

| | | | | |
|---|---|---|---|---|
| To_String | | _Type | String | Null_Value_Error |
| To_Unpadded_String | | _Not_Null | String | |
| To_Unpadded_String | | _Type | String | Null_Value_Error |
| To_SQL_Char_Not_Null | | String | _Not_Null | |
| To_SQL_Char | | String | _Type | |
| Unpadded_Length | | _Type | SQL_U_L[9] | Null_Value_Error |
| Substring[10] | | _Type | _Type | Constraint_Error |
| & | _Type | _Type | _Type | |

**Enumeration Domains**

| | | | | |
|---|---|---|---|---|
| Pred, Succ | | _Type | _Type | |
| Image | | _Type | SQL_Char | |
| Image | | _Not_Null | SQL_ChrNN | |
| Pos | | _Type | Integer | Null_Value_Error |
| Val | | Integer | _Type | |
| Value | | SQL_Char | _Type | |
| Value | | SQL_ChrNN | _Not_Null | |

**Boolean Functions**

| | | | | |
|---|---|---|---|---|
| not | | B_W_U | Boolean | |
| and, or, xor | B_W_U | B_W_U | Boolean | |
| To_Boolean | | B_W_U | Boolean | Null_Value_Error |
| Is_True, | B_W_U | B_W_U | Boolean | |
| Is_False, | B_W_U | B_W_U | Boolean | |
| Is_Unknown | B_W_U | B_W_U | Boolean | |

---

1. **"_Type" represents the type in the abstract domain of which objects that may be null are declared.**
2. **"_Not_Null" represents the type in the abstract domain of which objects that are not null may be declared.**
3. **"Assign" is a procedure. The result is returned in object "Left."**
4. **"B_W_U" is an abbreviation for Boolean_With_Unknown.**
5. **"SQL_ChrNN" is an abbreviation for SQL_Char_Not_Null.**
6. **"SQL_IntNN" is an abbreviation for SQL_Int_Not_Null.**
7. **"SQL_DblNN" is an abbreviation for SQL_Double_Precision_Not_Null.**
8. **"SQL_Dbl" is an abbreviation for SQL_Double_Precision.**
9. **"SQL_U_L" is an abbreviation for the SQL_Char_Pkg subtype SQL_Unpadded_Length.**
10. **Substring has two additional parameters: Start and Length, which are both of the SQL_Char_Pkg subtype SQL_Char_Length.**

## C.2 Standard Support Package Specifications

### C.2.1 SQL_Standard

The package SQL_Standard is defined in 12.4 of ISO/IEC 9075:1992 and is reproduced here for information only.

```
package SQL_Standard is
    package Character_Set renames csp;
    subtype Character_Type is Character_Set.cst;
    type Char is array (positive range <>) of Character_Type;
    type Bit is array (natural range <>) of boolean;
    type Smallint is range bs..ts;
    type Int is range bi..ti;
    type Real is digits dr;
    type Double_Precision is digits dd;
    subtype Indicator_Type is t;
    type Sqlcode_Type is range bsc..tsc;
    subtype Sql_Error is Sqlcode_Type range Sqlcode_Type'FIRST .. -1;
    subtype Not_Found is Sqlcode_Type range 100..100;
    type SQLSTATE_Type is new Char (1 .. 5);

-- csp is an implementor-defined package and cst is an
-- implementor-defined character type. bs, ts, bi, ti, dr, dd, bsc,
-- and tsc are implementor defined integral values. t is int or
-- smallint corresponding to an implementor-defined exact
-- numeric type of indicator parameters.

    package SQLSTATE_Codes is

         -- this subpackage contains a constant declaration for each
         -- SQLSTATE value defined by ISO/IEC 9075:1992. These constants
         -- are reproduced in the SAMeDL module (and therefore the Ada package)
         -- SAMeDL_Standard. See Annex A of this specification.
         -- The list of constants declared in 12.4 of ISO/IEC 9075:1992
         -- is definitive.

    end SQLSTATE_Codes;

end SQL_Standard;
```

### C.2.2 SQL_Boolean_Pkg

```
package SQL_Boolean_Pkg is

    type Boolean_with_Unknown is (FALSE, UNKNOWN, TRUE);

    ---- Three valued Logic operations --
    ---  three-val X three-val => three-val --
    function "not"  (Left : Boolean_with_Unknown)
        return Boolean_with_Unknown;
    pragma INLINE ("not");
    function "and" (Left, Right : Boolean_with_Unknown)
        return Boolean_with_Unknown;
    pragma INLINE ("and");
    function "or" (Left, Right : Boolean_with_Unknown)
        return Boolean_with_Unknown;
    pragma INLINE ("or");
    function "xor" (Left, Right : Boolean_with_Unknown)
        return Boolean_with_Unknown;
    pragma INLINE ("xor");

    --- three-val => bool or exception ---
    function To_Boolean (Left : Boolean_with_Unknown) return Boolean;
    pragma INLINE (To_Boolean);

    --- three-val => bool ---
    function Is_True (Left : Boolean_with_Unknown) return Boolean;
    pragma INLINE (Is_True);
    function Is_False (Left : Boolean_with_Unknown) return Boolean;
    pragma INLINE (Is_False);
    function Is_Unknown (Left : Boolean_with_Unknown) return Boolean;
    pragma INLINE (Is_Unknown);

end SQL_Boolean_Pkg;
```

## C.2.3 SQL_Int_Pkg

```
with SQL_Standard;
with SQL_Boolean_Pkg; use SQL_Boolean_Pkg;
with SQL_Char_Pkg; use SQL_Char_Pkg;
package SQL_Int_Pkg is

    type SQL_Int_Not_Null is new SQL_Standard.Int;

        ---- Possibly Null Integer ----

    type SQL_Int is limited private;

    function Null_SQL_Int return SQL_Int;
    pragma INLINE (Null_SQL_Int);

    -- this pair of functions convert between the
    --   null-bearing and non-null-bearing types.
    function Without_Null_Base(Value : SQL_Int)
        return SQL_Int_Not_Null;
    pragma INLINE (Without_Null_Base);
    -- With_Null_Base raises Null_Value_Error if the input
    --   value is null
    function With_Null_Base(Value : SQL_Int_Not_Null)
        return SQL_Int;
    pragma INLINE (With_Null_Base);

    -- this procedure implements range checking
    -- note: it is not meant to be used directly
    --   by application programmers
    -- see the generic package SQL_Int_Ops
    -- raises constraint_error if not
    --   (First <= Right <= Last)
    procedure Assign_with_check (
        Left : in out SQL_Int; Right : SQL_Int;
        First, Last : SQL_Int_Not_Null);
    pragma INLINE (Assign_with_check);

    -- the following functions implement three valued
    --   arithmetic
    -- if either input to any of these functions is null
    --   the function returns the null value; otherwise
    --   they perform the indicated operation
    -- these functions raise no exceptions
    function "+"(Right : SQL_Int) return SQL_Int;
    pragma INLINE ("+");
    function "-"(Right : SQL_Int) return SQL_Int;
    pragma INLINE ("-");
    function "abs"(Right : SQL_Int) return SQL_Int;
    pragma INLINE ("abs");
    function "+"(Left, Right : SQL_Int) return SQL_Int;
    pragma INLINE ("+");
    function "*"(Left, Right : SQL_Int) return SQL_Int;
    pragma INLINE ("*");
    function "-"(Left, Right : SQL_Int) return SQL_Int;
    pragma INLINE ("-");
    function "/"(Left, Right : SQL_Int) return SQL_Int;
    pragma INLINE ("/");
    function "mod" (Left, Right : SQL_Int) return SQL_Int;
    pragma INLINE ("mod");
    function "rem" (Left, Right : SQL_Int) return SQL_Int;
```

```
pragma INLINE ("rem");
function "**" (Left : SQL_Int; Right: Integer) return SQL_Int;
pragma INLINE ("**");

-- simulation of 'IMAGE and 'VALUE that
-- return/take SQL_Char[_Not_Null] instead of string
function IMAGE (Left : SQL_Int_Not_Null) return SQL_Char_Not_Null;
function IMAGE (Left : SQL_Int) return SQL_Char;
pragma INLINE (IMAGE);
function VALUE (Left : SQL_Char_Not_Null) return SQL_Int_Not_Null;
function VALUE (Left : SQL_Char) return SQL_Int;
pragma INLINE (VALUE);

    -- Logical Operations --
    -- type X type => Boolean_with_unknown --
-- these functions implement three valued logic
-- if either input is the null value, the functions
--   return the truth value UNKNOWN; otherwise they
--   perform the indicated comparison.
-- these functions raise no exceptions
function Equals (Left, Right : SQL_Int)
    return Boolean_with_Unknown;
pragma INLINE (Equals);
function Not_Equals (Left, Right : SQL_Int)
    return Boolean_with_Unknown;
pragma INLINE (Not_Equals);
function "<" (Left, Right : SQL_Int) return Boolean_with_Unknown;
pragma INLINE ("<");
function ">" (Left, Right : SQL_Int) return Boolean_with_Unknown;
pragma INLINE (">");
function "<=" (Left, Right : SQL_Int) return Boolean_with_Unknown;
pragma INLINE ("<=");
function ">=" (Left, Right : SQL_Int) return Boolean_with_Unknown;
pragma INLINE (">=");

    -- type => Boolean --
function Is_Null(Value : SQL_Int) return Boolean;
pragma INLINE (Is_Null);
function Not_Null(Value : SQL_Int) return Boolean;
pragma INLINE (Not_Null);

-- These functions of class type => Boolean
-- equate UNKNOWN with FALSE. That is, they return TRUE
-- only when the function returns TRUE. UNKNOWN and FALSE
-- are mapped to FALSE.
function "=" (Left, Right : SQL_Int) return Boolean;
pragma INLINE ("=");
function "<" (Left, Right : SQL_Int) return Boolean;
pragma INLINE ("<");
function ">" (Left, Right : SQL_Int) return Boolean;
pragma INLINE (">");
function "<=" (Left, Right : SQL_Int) return Boolean;
pragma INLINE ("<=");
function ">=" (Left, Right : SQL_Int) return Boolean;
pragma INLINE (">=");

-- this generic is instantiated once for every abstract
--   domain based on the SQL type Int.
-- the three subprogram formal parameters are meant to
--   default to the programs declared above.
-- that is, the package should be instantiated in the
```

```
    --   scope of a use clause for SQL_Int_Pkg.
    -- the two actual types together form the abstract
    --   domain.
    -- the purpose of the generic is to create functions
    --   which convert between the two actual types and a
    --   procedure which implements a range constrained
    --   assignment for the null-bearing type.
    -- the bodies of these subprograms are calls to
    --   subprograms declared above and passed as defaults to
    --   the generic.
    generic
        type With_Null_type is limited private;
        type Without_null_type is range <>;
        with function With_Null_Base(Value : SQL_Int_Not_Null)
            return With_Null_Type is <>;
        with function Without_Null_Base(Value : With_Null_Type)
            return  SQL_Int_Not_Null is <>;
        with procedure Assign_with_check (
            Left : in out With_Null_Type; Right : With_Null_Type;
            First, Last : SQL_Int_Not_Null) is <>;
    package SQL_Int_Ops is
        function With_Null (Value : Without_Null_type)
            return With_Null_type;
        pragma INLINE (With_Null);
        function Without_Null (Value : With_Null_Type)
            return Without_Null_type;
        pragma INLINE (Without_Null);
        procedure assign (
            Left  : in out With_null_Type;
            Right : in With_null_type);
        pragma INLINE (assign);
    end SQL_Int_Ops;

private

    type SQL_Int is record
        Is_Null: Boolean := True;
        Value: SQL_Int_Not_Null;
    end record;

end SQL_Int_Pkg;
```

### C.2.4 SQL_Smallint_Pkg

```
with SQL_Standard;
with SQL_Boolean_Pkg; use SQL_Boolean_Pkg;
with SQL_Char_Pkg; use SQL_Char_Pkg;
package SQL_Smallint_Pkg is

    type SQL_Smallint_Not_Null is new SQL_Standard.Smallint;

        ---- Possibly Null Integer ----

    type SQL_Smallint is limited private;

    function Null_SQL_Smallint return SQL_Smallint;
    pragma INLINE (Null_SQL_Smallint);

    -- this pair of functions converts between the
    --   null-bearing and not null-bearing types.
    function Without_Null_Base(Value : SQL_Smallint)
        return SQL_Smallint_Not_Null;
    pragma INLINE (Without_Null_Base);
    -- With_Null_Base raises Null_Value_Error if the input
    --   value is null
    function With_Null_Base(Value : SQL_Smallint_Not_Null)
        return SQL_Smallint;
    pragma INLINE (With_Null_Base);

    -- this procedure implements range checking
    -- note: it is not meant to be used directly
    --   by application programmers
    -- see the generic package SQL_Smallint_Op
    -- raises constraint_error if not
    --   (First <= Right <= Last)
    procedure Assign_with_check (
        Left : in out SQL_Smallint; Right : SQL_Smallint;
        First, Last : SQL_Smallint_Not_Null);
    pragma INLINE (Assign_with_check);

    -- the following functions implement three valued
    --   arithmetic
    -- if either input to any of these functions is null
    --   the function returns the null value; otherwise
    --   they perform the indicated operation
    -- these functions raise no exceptions
    function "+"(Right : SQL_Smallint) return SQL_Smallint;
    pragma INLINE ("+");
    function "-"(Right : SQL_Smallint) return SQL_Smallint;
    pragma INLINE ("-");
    function "abs"(Right : SQL_Smallint) return SQL_Smallint;
    pragma INLINE ("abs");
    function "+"(Left, Right : SQL_Smallint) return SQL_Smallint;
    pragma INLINE ("+");
    function "*"(Left, Right : SQL_Smallint) return SQL_Smallint;
    pragma INLINE ("*");
    function "-"(Left, Right : SQL_Smallint) return SQL_Smallint;
    pragma INLINE ("-");
    function "/"(Left, Right : SQL_Smallint) return SQL_Smallint;
    pragma INLINE ("/");
    function "mod" (Left, Right : SQL_Smallint) return SQL_Smallint;
    pragma INLINE ("mod");
    function "rem" (Left, Right : SQL_Smallint) return SQL_Smallint;
```

```
pragma INLINE ("rem");
function "**" (Left : SQL_Smallint; Right: Integer)
    return SQL_Smallint;
pragma INLINE ("**");

-- simulation of 'IMAGE and 'VALUE that
-- return/take SQL_Char[_Not_Null] instead of string
function IMAGE (Left : SQL_Smallint_Not_Null)
    return SQL_Char_Not_Null;
function IMAGE (Left : SQL_Smallint) return SQL_Char;
pragma INLINE (IMAGE);
function VALUE (Left : SQL_Char_Not_Null)
    return SQL_Smallint_Not_Null;
function VALUE (Left : SQL_Char) return SQL_Smallint;
pragma INLINE (VALUE);

    -- Logical Operations --
    -- type X type => Boolean_with_unknown --
-- these functions implement three valued logic
-- if either input is the null value, the functions
--   return the truth value UNKNOWN; otherwise they
--   perform the indicated comparison.
-- these functions raise no exceptions
function Equals (Left, Right : SQL_Smallint)
    return Boolean_with_Unknown;
pragma INLINE (Equals);
function Not_Equals (Left, Right : SQL_Smallint)
    return  Boolean_with_Unknown;
pragma INLINE (Not_Equals);
function "<" (Left, Right : SQL_Smallint)
    return Boolean_with_Unknown;
pragma INLINE ("<");
function ">" (Left, Right : SQL_Smallint)
    return Boolean_with_Unknown;
pragma INLINE (">");
function "<=" (Left, Right : SQL_Smallint)
    return Boolean_with_Unknown;
pragma INLINE ("<=");
function ">=" (Left, Right : SQL_Smallint)
    return Boolean_with_Unknown;
pragma INLINE (">=");

    -- type => Boolean --
function Is_Null(Value : SQL_Smallint) return Boolean;
pragma INLINE (Is_Null);
function Not_Null(Value : SQL_Smallint) return Boolean;
pragma INLINE (Not_Null);

-- These functions of class type => Boolean
-- equate UNKNOWN with FALSE. That is, they return TRUE
-- only when the function returns TRUE. UNKNOWN and FALSE
-- are mapped to FALSE.
function "=" (Left, Right : SQL_Smallint) return Boolean;
pragma INLINE ("=");
function "<" (Left, Right : SQL_Smallint) return Boolean;
pragma INLINE ("<");
function ">" (Left, Right : SQL_Smallint) return Boolean;
pragma INLINE (">");
function "<=" (Left, Right : SQL_Smallint) return Boolean;
pragma INLINE ("<=");
function ">=" (Left, Right : SQL_Smallint) return Boolean;
```

```
    pragma INLINE (">=");

    -- this generic is instantiated once for every abstract
    --   domain based on the SQL type Smallint.
    -- the three subprogram formal parameters are meant to
    --   default to the programs declared above.
    -- that is, the package should be instantiated in the
    --   scope of a use clause for SQL_Smallint_Pkg.
    -- the two actual types together form the abstract
    --   domain.
    -- the purpose of the generic is to create functions
    --   which convert between the two actual types and a
    --   procedure which implements a range constrained
    --   assignment for the null-bearing type.
    -- the bodies of these subprograms are calls to
    --   subprograms declared above and passed as defaults to
    --   the generic.
    generic
        type With_Null_type is limited private;
        type Without_null_type is range <>;
        with function With_Null_Base(Value : SQL_Smallint_Not_Null)
            return With_Null_Type is <>;
        with function Without_Null_Base(Value : With_Null_Type)
            return  SQL_Smallint_Not_Null is <>;
        with procedure Assign_with_check (
            Left : in out With_Null_Type; Right : With_Null_Type;
            First, Last : SQL_Smallint_Not_Null) is <>;
    package SQL_Smallint_Ops is
        function With_Null (Value : Without_Null_type)
            return With_Null_type;
        pragma INLINE (With_Null);
        function Without_Null (Value : With_Null_Type)
            return Without_Null_type;
        pragma INLINE (Without_Null);
        procedure assign (
            Left  : in out With_null_Type;
            Right : in With_null_type);
        pragma INLINE (assign);
    end SQL_Smallint_Ops;

private

    type SQL_Smallint is record
        Is_Null: Boolean := True;
        Value: SQL_Smallint_Not_Null;
    end record;

end SQL_Smallint_Pkg;
```

### C.2.5 SQL_Real_Pkg

```
with SQL_Standard;
with SQL_Boolean_Pkg; use SQL_Boolean_Pkg;
package SQL_Real_Pkg is

    type SQL_Real_Not_Null is new SQL_Standard.Real;

        ---- Possibly Null Real ----

    type SQL_Real is limited private;

    function Null_SQL_Real return SQL_Real;
    pragma INLINE (Null_SQL_Real);

    -- this pair of functions converts between the
    --   null-bearing and not null-bearing types
    function Without_Null_Base(Value : SQL_Real)
        return SQL_Real_Not_Null;
    pragma INLINE (Without_Null_Base);
    -- With_Null_Base raises Null_Value_Error if the input
    --   value is null
    function With_Null_Base(Value : SQL_Real_Not_Null)
        return SQL_Real;
    pragma INLINE (With_Null_Base);

    -- this procedure implements range checking
    -- note: it is not meant to be used directly
    --   by application programmers
    -- see the generic package SQL_Real_Ops
    -- raises constraint_error if not
    --   (First <= Right <= Last)
    procedure Assign_with_Check  (
        Left : in out SQL_Real; Right : SQL_Real;
        First, Last : SQL_Real_Not_Null);
    pragma INLINE (Assign_with_Check);

    -- the following functions implement three valued
    -- arithmetic
    -- if either input to any of these functions is null
    --   the function returns the null value; otherwise
    --   they perform the indicated operation
    -- these functions raise no exceptions
    function "+"(Right : SQL_Real) return SQL_Real;
    pragma INLINE ("+");
    function "-"(Right : SQL_Real) return SQL_Real;
    pragma INLINE ("-");
    function "abs"(Right : SQL_Real) return SQL_Real;
    pragma INLINE ("abs");
    function "+"(Left, Right : SQL_Real) return SQL_Real;
    pragma INLINE ("+");
    function "*"(Left, Right : SQL_Real) return SQL_Real;
    pragma INLINE ("*");
    function "-"(Left, Right : SQL_Real) return SQL_Real;
    pragma INLINE ("-");
    function "/"(Left, Right : SQL_Real) return SQL_Real;
    pragma INLINE ("/");
    function "**"(Left : SQL_Real; Right : Integer) return SQL_Real;
    pragma INLINE ("**");

        -- Logical Operations --
```

```
    -- type X type => Boolean_with_unknown --
-- these functions implement three valued logic
-- if either input is the null value, the functions
--   return the truth value UNKNOWN;  otherwise they
--   perform the indicated comparison.
-- these functions raise no exceptions
function Equals (Left, Right : SQL_Real)
    return Boolean_with_Unknown;
pragma INLINE (Equals);
function Not_Equals (Left, Right : SQL_Real)
    return  Boolean_with_Unknown;
pragma INLINE (Not_Equals);
function "<" (Left, Right : SQL_Real) return Boolean_with_Unknown;
pragma INLINE ("<");
function ">" (Left, Right : SQL_Real) return Boolean_with_Unknown;
pragma INLINE (">");
function "<=" (Left, Right : SQL_Real) return Boolean_with_Unknown;
pragma INLINE ("<=");
function ">=" (Left, Right : SQL_Real) return Boolean_with_Unknown;
pragma INLINE (">=");

    -- type => Boolean --
function Is_Null(Value : SQL_Real) return Boolean;
pragma INLINE (Is_Null);
function Not_Null(Value : SQL_Real) return Boolean;
pragma INLINE (Not_Null);

-- These functions of class type => Boolean
-- equate UNKNOWN with FALSE. That is, they return TRUE
-- only when the function returns TRUE. UNKNOWN and FALSE
-- are mapped to FALSE.
function "=" (Left, Right : SQL_Real) return Boolean;
pragma INLINE ("=");
function "<" (Left, Right : SQL_Real) return Boolean;
pragma INLINE ("<");
function ">" (Left, Right : SQL_Real) return Boolean;
pragma INLINE (">");
function "<=" (Left, Right : SQL_Real) return Boolean;
pragma INLINE ("<=");
function ">=" (Left, Right : SQL_Real) return Boolean;
pragma INLINE (">=");

-- this generic is instantiated once for every abstract
--   domain based on the SQL type Real.
-- the three subprogram formal parameters are meant to
--   default to the programs declared above.
-- that is, the package should be instantiated in the
--   scope of the use clause for SQL_Real_Pkg.
-- the two actual types together form the abstract
--   domain.
-- the purpose of the generic is to create functions
--   which convert between the two actual types and a
--   procedure which implements a range constrained
--   assignment for the null-bearing type.
-- the bodies of these subprograms are calls to
--   subprograms declared above and passed as defaults to
--   the generic.
generic
    type With_Null_type is limited private;
    type Without_null_type is digits <>;
    with function With_Null_Base(Value : SQL_Real_Not_Null)
```

```
                return With_Null_Type is <>;
            with function Without_Null_Base(Value : With_Null_Type)
                return  SQL_Real_Not_Null is <>;
            with procedure Assign_with_check (
                Left : in out With_Null_Type; Right : With_Null_Type;
                First, Last : SQL_Real_Not_Null) is <>;
        package SQL_Real_Ops is
            function With_Null (Value : Without_Null_type)
                return With_Null_type;
            pragma INLINE (With_Null);
            function Without_Null (Value : With_Null_Type)
                return Without_Null_type;
            pragma INLINE (Without_Null);
            procedure assign (
                Left  : in out With_Null_Type;
                Right : in With_Null_type);
            pragma INLINE (assign);
        end SQL_Real_Ops;

    private

        type SQL_Real is record
            Is_Null: Boolean := True;
            Value: SQL_Real_Not_Null;
        end record;

    end SQL_Real_Pkg;
```

### C.2.6 SQL_Double_Precision_Pkg

```
with SQL_Standard;
with SQL_Boolean_Pkg; use SQL_Boolean_Pkg;
package SQL_Double_Precision_Pkg is

    type SQL_Double_Precision_Not_Null is new
        SQL_Standard.Double_Precision;

        ---- Possibly Null Double_Precision ----

    type SQL_Double_Precision is limited private;

    function Null_SQL_Double_Precision return SQL_Double_Precision;
    pragma INLINE (Null_SQL_Double_Precision);

    -- this pair of functions converts between the
    --   null-bearing and not null-bearing types.
    function Without_Null_Base(Value : SQL_Double_Precision)
        return SQL_Double_Precision_Not_Null;
    pragma INLINE (Without_Null_Base);
    -- With_Null_Base raises Null_Value_Error if the input
    --   value is null
    function With_Null_Base(Value : SQL_Double_Precision_Not_Null)
        return SQL_Double_Precision;
    pragma INLINE (With_Null_Base);

    -- this procedure implements range checking
    -- note: it is not meant to be used directly
    --   by application programmers
    -- see the generic package SQL_Double_Precision_Op
    -- raises constraint_error if not
    --   (First <= Right <= Last)
    procedure Assign_with_Check  (
        Left : in out SQL_Double_Precision;
        Right : SQL_Double_Precision;
        First, Last : SQL_Double_Precision_Not_Null);
    pragma INLINE (Assign_with_Check);

    -- the following functions implement three valued
    --   arithmetic
    -- if either input to any of these functions is null
    --   the function returns the null value; otherwise they
    --   perform the indicated operation
    -- these functions raise no exceptions
    function "+"(Right : SQL_Double_Precision)
        return SQL_Double_Precision;
    pragma INLINE ("+");
    function "-"(Right : SQL_Double_Precision)
        return SQL_Double_Precision;
    pragma INLINE ("-");
    function "abs"(Right : SQL_Double_Precision)
        return SQL_Double_Precision;
    pragma INLINE ("abs");
    function "+"(Left, Right : SQL_Double_Precision)
        return SQL_Double_Precision;
    pragma INLINE ("+");
    function "*"(Left, Right : SQL_Double_Precision)
        return SQL_Double_Precision;
    pragma INLINE ("*");
    function "-"(Left, Right : SQL_Double_Precision)
```

```
        return SQL_Double_Precision;
    pragma INLINE ("-");
    function "/"(Left, Right : SQL_Double_Precision)
        return SQL_Double_Precision;
    pragma INLINE ("/");
    function "**"(Left : SQL_Double_Precision; Right : Integer)
        return SQL_Double_Precision;
    pragma INLINE ("**");

        -- Logical Operations --
        -- type X type => Boolean_with_unknown --
    -- these functions implement three valued logic
    -- if either input is the null value, the functions
    --   return the truth value UNKNOWN; otherwise they
    --   perform the indicated comparison.
    -- these functions raise no exceptions
    function Equals (Left, Right : SQL_Double_Precision)
        return Boolean_with_Unknown;
    pragma INLINE (Equals);
    function Not_Equals (Left, Right : SQL_Double_Precision)
        return Boolean_with_Unknown;
    pragma INLINE (Not_Equals);
    function "<" (Left, Right : SQL_Double_Precision)
        return Boolean_with_Unknown;
    pragma INLINE ("<");
    function ">" (Left, Right : SQL_Double_Precision)
        return Boolean_with_Unknown;
    pragma INLINE (">");
    function "<=" (Left, Right : SQL_Double_Precision)
        return Boolean_with_Unknown;
    pragma INLINE ("<=");
    function ">=" (Left, Right : SQL_Double_Precision)
        return Boolean_with_Unknown;
    pragma INLINE (">=");

        -- type => Boolean --
    function Is_Null(Value : SQL_Double_Precision) return Boolean;
    pragma INLINE (Is_Null);
    function Not_Null(Value : SQL_Double_Precision) return Boolean;
    pragma INLINE (Not_Null);

    -- These functions of class type => Boolean
    -- equate UNKNOWN with FALSE. That is, they return TRUE
    -- only when the function returns TRUE. UNKNOWN and FALSE
    -- are mapped to FALSE.
    function "=" (Left, Right : SQL_Double_Precision) return Boolean;
    function "<" (Left, Right : SQL_Double_Precision) return Boolean;
    function ">" (Left, Right : SQL_Double_Precision) return Boolean;
    function "<=" (Left, Right : SQL_Double_Precision) return Boolean;
    function ">=" (Left, Right : SQL_Double_Precision) return Boolean;

    -- this generic is instantiated once for every abstract
    --   domain based on the SQL type Double_Precision.
    -- the three subprogram formal parameters are meant to
    --   default to the programs declared above.
    -- that is, the package should be instantiated in the
    --   scope of the use clause for
    --   SQL_Double_Precision_Pkg.
    -- the two actual types together form the abstract
    --   domain.
    -- the purpose of the generic is to create functions
```

```
    --   which convert between the two actual types and a
    --   procedure which implements a range constrained
    --   assignment for the null-bearing type.
    -- the bodies of these subprograms are calls to
    --   subprograms declared above and passed as defaults
    --   to the generic.
    generic
        type With_Null_type is limited private;
        type Without_null_type is digits <>;
        with function With_Null_Base(
            Value : SQL_Double_Precision_Not_Null)
            return With_Null_Type is <>;
        with function Without_Null_Base(Value : With_Null_Type)
            return SQL_Double_Precision_Not_Null is <>;
        with procedure Assign_with_check (
            Left : in out With_Null_Type; Right : With_Null_Type;
            First, Last : SQL_Double_Precision_Not_Null) is <>;
    package SQL_Double_Precision_Ops is
        function With_Null (Value : Without_Null_type)
            return With_Null_type;
        pragma INLINE (With_Null);
        function Without_Null (Value : With_Null_Type)
            return Without_Null_type;
        pragma INLINE (Without_Null);
        procedure assign (
            Left  : in out With_null_Type;
            Right : in With_null_type);
        pragma INLINE (assign);
    end SQL_Double_Precision_Ops;

private

    type SQL_Double_Precision is record
        Is_Null: Boolean := True;
        Value: SQL_Double_Precision_Not_Null;
    end record;

end SQL_Double_Precision_Pkg;
```

### C.2.7 SQL_Char_Pkg

```
with SQL_System; use SQL_System;
with SQL_Boolean_Pkg; use SQL_Boolean_Pkg;
with SQL_Standard;
package SQL_Char_Pkg is

    subtype SQL_Char_Length is natural
        range 1 .. MAXCHRLEN;
    subtype SQL_Unpadded_Length is natural
        range 0 .. MAXCHRLEN;

    type SQL_Char_Not_Null is new SQL_Standard.Char;

    type SQL_Char(Length : SQL_Char_Length) is limited private;

    function Null_SQL_Char return SQL_Char;
    pragma INLINE (Null_SQL_Char);

    -- the next three functions convert between
    --   null-bearing and not null-bearing-types
    -- Without_Null_Base and With_Null_Base are
    --   inverses (mod. null values)
    -- see also SQL_Char_Ops generic package below
    function With_Null_Base(Value : SQL_Char_Not_Null)
        return SQL_Char;
    pragma INLINE (With_Null_Base);
    -- Without_Null_Base and Without_Null_Base_Unpadded raise
    --   null_value_error on the null input
    function Without_Null_Base(Value : SQL_Char)
        return SQL_Char_Not_Null;
    pragma INLINE (Without_Null_Base);
    -- Without_Null_Unpadded_Base removes trailing blanks from
    --   the input
    function Without_Null_Unpadded_Base(Value : SQL_Char)
        return SQL_Char_Not_Null;
    pragma INLINE (Without_Null_Unpadded_Base);
    -- axiom: unpadded_Length(x) =
    --   Without_Null_Unpadded_Base(x)'Length
    -- both functions raise null_value_error if x is null

    -- the next six functions convert between Standard.String
    --   types and the SQL_Char and SQL_Char_Not_Null types
    function To_String (Value : SQL_Char_Not_Null)
        return String;
    function To_String (Value : SQL_Char)
        return String;
    function To_Unpadded_String (Value : SQL_Char_Not_Null)
        return String;
    function To_Unpadded_String (Value : SQL_Char)
        return String;
    pragma INLINE (To_Unpadded_String);
    -- this INLINE works for BOTH functions!!
    function To_SQL_Char_Not_Null (Value : String)
        return SQL_Char_Not_Null;
    function To_SQL_Char (Value : String)
        return SQL_Char;
    pragma INLINE (To_SQL_Char);

    function Unpadded_Length (Value : SQL_Char)
        return SQL_Unpadded_Length;
```

```
pragma INLINE (Unpadded_Length);

procedure Assign(
    Left : out SQL_Char;
    Right : SQL_Char);
pragma INLINE (Assign);

-- Substring(x,k,m) returns the substring of x starting
--   at position k (relative to 1) with length m.
-- returns null value if x is null
-- raises constraint_error if Start < 1 or Length < 1 or
--   Start + Length - 1 > x.Length
function Substring (
    Value : SQL_Char;
    Start, Length : SQL_Char_Length)
    return  SQL_Char;
pragma INLINE (Substring);

-- "&" returns null if either parameter is null;
--   otherwise performs concatenation in the usual way,
--   preserving all blanks.
-- may raise constraint_error implicitly if result is
--   too large (i.e., greater than SQL_Char_Length'Last
function "&" (Left, Right : SQL_Char)
    return SQL_Char;
pragma INLINE ("&");

    -- Logical Operations --
    -- type X type => Boolean_with_unknown --
-- the comparison operators return the boolean value
--   UNKNOWN if either parameter is null; otherwise,
--   the comparison is done in accordance with
--   ANSI X3.135-1986 para 5.11 general rule 5; that is,
--   the shorter of the two string parameters is
--   effectively padded with blanks to be the length of
--   the longer string and a standard Ada comparison is
--   then made
function Equals (Left, Right : SQL_Char)
    return Boolean_with_Unknown;
pragma INLINE (Equals);
function Not_Equals (Left, Right : SQL_Char)
    return Boolean_with_Unknown;
pragma INLINE (Not_Equals);
function "<" (Left, Right : SQL_Char) return Boolean_with_Unknown;
pragma INLINE ("<");
function ">" (Left, Right : SQL_Char) return Boolean_with_Unknown;
pragma INLINE (">");
function "<=" (Left, Right : SQL_Char) return Boolean_with_Unknown;
pragma INLINE ("<=");
function ">=" (Left, Right : SQL_Char) return Boolean_with_Unknown;
pragma INLINE (">=");

    -- type => Boolean --
function Is_Null(Value : SQL_Char) return Boolean;
pragma INLINE (Is_Null);
function Not_Null(Value : SQL_Char) return Boolean;
pragma INLINE (Not_Null);

-- These functions of class type => Boolean
-- equate UNKNOWN with FALSE. That is, they return TRUE
-- only when the function returns TRUE. UNKNOWN and FALSE
```

```
    -- are mapped to FALSE.
    function "=" (Left, Right : SQL_Char) return Boolean;
    pragma INLINE ("=");
    function "<" (Left, Right : SQL_Char) return Boolean;
    pragma INLINE ("<");
    function ">" (Left, Right : SQL_Char) return Boolean;
    pragma INLINE (">");
    function "<=" (Left, Right : SQL_Char) return Boolean;
    pragma INLINE ("<=");
    function ">=" (Left, Right : SQL_Char) return Boolean;
    pragma INLINE (">=");

    -- the purpose of the following generic is to generate
    --   conversion functions between a type derived from
    --   SQL_Char_Not_Null, which are effectively Ada
    --   strings and a type derived from SQL_Char, which
    --   mimic the behavior of SQL strings.
    -- the subprogram formals are meant to default; that is,
    --   this generic should be instantiated in the scope
    --   of an use clause for SQL_Char_Pkg.
    generic
        type With_Null_Type is limited private;
        type Without_Null_Type is array (positive range <>)
            of SQL_Standard.Character_Type;
        with function With_Null_Base (Value: SQL_Char_Not_Null)
            return With_Null_Type is <>;
        with function Without_Null_Base (Value: With_Null_Type)
            return SQL_Char_Not_Null is <>;
        with function Without_Null_Unpadded_Base (Value: With_Null_Type)
            return SQL_Char_Not_Null is <>;
    package SQL_Char_Ops is
        function With_Null (Value : Without_Null_Type)
            return With_Null_Type;
        pragma INLINE (With_Null);
        function Without_Null (Value : With_Null_Type)
            return Without_Null_Type;
        pragma INLINE (Without_Null);
        function Without_Null_Unpadded (Value : With_Null_Type)
            return Without_Null_Type;
        pragma INLINE (Without_Null_Unpadded);
    end SQL_Char_Ops;

private

    type SQL_Char(Length : SQL_Char_Length) is record
        Is_Null: Boolean := True;
        Unpadded_Length: SQL_Unpadded_Length;
        Text: SQL_Char_Not_Null(1 .. Length);
    end record;

end SQL_Char_Pkg;
```

### C.2.8 SQL_Enumeration_Pkg

```
with SQL_Boolean_Pkg; use SQL_Boolean_Pkg;
with SQL_Char_Pkg; use SQL_Char_Pkg;
generic
    type SQL_Enumeration_Not_Null is (<>);
package SQL_Enumeration_Pkg is

        ---- Possibly Null Enumeration ----

    type SQL_Enumeration is limited private;

    function Null_SQL_Enumeration return SQL_Enumeration;
    pragma INLINE (Null_SQL_Enumeration);

    -- this pair of functions convert between the
    --   null-bearing and not null-bearing types.
    function Without_Null(Value : in SQL_Enumeration)
        return SQL_Enumeration_Not_Null;
    pragma INLINE (Without_Null);
    -- With_Null raises Null_Value_Error if the input
    --   value is null
    function With_Null(Value : in SQL_Enumeration_Not_Null)
        return SQL_Enumeration;
    pragma INLINE (With_Null);

    procedure Assign (
        Left : in out SQL_Enumeration;
        Right : in SQL_Enumeration);
    pragma INLINE (Assign);

        -- Logical Operations --
        -- type X type => Boolean_with_unknown --
    -- these functions implement three valued logic
    -- if either input is the null value, the functions
    --   return the truth value UNKNOWN; otherwise they
    --   perform the indicated comparison.
    -- these functions raise no exceptions
    function Equals (Left, Right : SQL_Enumeration)
        return Boolean_with_Unknown;
    function Not_Equals (Left, Right : SQL_Enumeration)
        return Boolean_with_Unknown;
    pragma INLINE (Not_Equals);
    function "<" (Left, Right : SQL_Enumeration)
        return Boolean_with_Unknown;
    function ">" (Left, Right : SQL_Enumeration)
        return Boolean_with_Unknown;
    function "<=" (Left, Right : SQL_Enumeration)
        return Boolean_with_Unknown;
    function ">=" (Left, Right : SQL_Enumeration)
        return Boolean_with_Unknown;

        -- type => Boolean --
    function Is_Null (Value : SQL_Enumeration) return Boolean;
    pragma INLINE (Is_Null);
    function Not_Null (Value : SQL_Enumeration) return Boolean;
    pragma INLINE (Not_Null);
    function "=" (Left, Right : SQL_Enumeration) return Boolean;
    pragma INLINE ("=");
    function "<" (Left, Right : SQL_Enumeration) return Boolean;
    pragma INLINE ("<");
```

```
    function ">" (Left, Right : SQL_Enumeration) return Boolean;
    pragma INLINE (">");
    function "<=" (Left, Right : SQL_Enumeration) return Boolean;
    pragma INLINE ("<=");
    function ">=" (Left, Right : SQL_Enumeration) return Boolean;
    pragma INLINE (">=");

    -- the following six functions mimic the
    --   'Pred, 'Succ, 'Image, 'Pos, 'Val, and 'Value
    --   attributes of the SQL_Enumeration_Not_Null type, passed
    --   in, for the associated SQL_Enumeration (null) type
    -- they all raise the Null_Value_Error exception if a null
    --   value is passed in
    -- Pred raises the Constraint_Error exception if the value
    --   passed in is equal to SQL_Enumeration_Not_Null'Last
    -- Succ raises the Constraint_Error exception if the value
    --   passed in is equal to SQL_Enumeration_Not_Null'First
    -- Val raises the Constraint_Error exception if the value passed
    --   in is not in the range P'POS(P'FIRST)..P'POS(P'LAST) for
    --   type P
    -- Value raises the Constraint_Error exception if the sequence of
    --   characters passed in does not have the syntax of an
    --   enumeration literal for the instantiated enumeration type
    function Pred (Value : in SQL_Enumeration)
        return SQL_Enumeration;
    pragma INLINE (Pred);
    function Succ (Value : in SQL_Enumeration)
        return SQL_Enumeration;
    pragma INLINE (Succ);
    function Pos (Value : in SQL_Enumeration) return Integer;
    pragma INLINE (Pos);
    function Image (Value : in SQL_Enumeration) return SQL_Char;
    function Image (Value : in SQL_Enumeration_Not_Null)
        return  SQL_Char_Not_Null;
    pragma INLINE (Image);
    function Val (Value : in Integer) return SQL_Enumeration;
    pragma INLINE (Val);
    function Value (Value : in SQL_Char) return SQL_Enumeration;
    function Value (Value : in SQL_Char_Not_Null)
        return  SQL_Enumeration_Not_Null;
    pragma INLINE (Value);

private

    type SQL_Enumeration is record
        Is_Null: Boolean := True;
        Value: SQL_Enumeration_Not_Null;
    end record;

end SQL_Enumeration_Pkg;
```

# Annex D: Transform Chart

.

.

.

# Annex E: Glossary

**Abstract interface.** A set of Ada package specifications containing the type and procedure declarations to be used by an Ada application program to access the database.

**Abstract module.** A module that specifies the database routines needed by an Ada application.

**Assignment context.** A value expression appears in an assignment context if the value of that value expression is to be implicitly or explicitly assigned to an object. The assignment contexts are: select parameters, constant declarations, values in a VALUES list of an insert statement, set items in an update statement.

**Base domain.** A template for defining domains.

**Conform.** A value expression in an assignment context conforms to a target domain if the rules of SQL allow the assignment of a value of the data class of the expression to an object of the data class of the domain.

**Conversion method.** A method of converting non-null data between objects of the not null-bearing type, the null-bearing type, and the database type associated with the domain.

**Correlation name.** See ISO/IEC 9075:1992.

**Cursor.** See ISO/IEC 9075:1992.

**Data class.** The data class of a value is either character, integer, fixed, float, or enumeration. The data class of a domain determines which values may be converted, implicitly or explicitly, to the domain.

**Database type.** The SQL data type to be used with an object of that domain when it appears in an SQL parameter declaration. This need not be the same as the type of the data as is stored in the database.

**Definitional module.** A module that contains shared definitions: that is, declarations of base domains, domains, subdomains, constants, records, exceptions, enumerations, and status maps that are used by other modules.

**Domain.** The set of values and applicable operations for objects associated with a domain. A domain is similar to an Ada type.

**Exposed.** The exposed name of a module (table or view) that appears in a context clause (table ref) containing an as phrase (correlation name) is the identifier in the associated as phrase (correlation name); the module name (table name) is hidden. If the as phrase (correlation name) is not present, the exposed name is that module’s (table’s or view’s) name. The exposed name of a table or module is the name by which that table or module is referenced.

**Extended.** A table, view, module, procedure, cursor, or cursor procedure that includes some nonstandard operation or feature.

**Hidden.** See exposed.

**Module.** A definitional module, a schema module, or an abstract module.

**Not null type.** The Ada type associated with objects of a domain that may not take a null value.

**Null type.** The Ada type associated with objects of a domain that may take a null value.

**Null value.** SQL's means of recording missing information. A null value in a column indicates that nothing is known about the value that should occupy the column.

**Options.** The aspects of the base domain that are essential to the declaration of domains based upon the base domain. In particular they define the base domain's null- and not null-bearing type name, data class, database type, and conversion methods.

**Patterns.** A template used to create the Ada constructs that implement the Ada semantics of a domain, subdomain, or derived domain declaration.

**Row record.** The Ada record associated with procedures that contain either a fetch, select, insert statement. It is used to transmit the database data to or from the client program.

**Row record type.** The Ada type of the row record.

**SAME.** SQL Ada Module Extensions.

**SAMeDL.** SQL Ada Module Description Language.

**Schema module.** A module containing definitions of database tables and views.

**SQL.** Formerly an acronym for Structured Query Language, SQL is the name of the relational database language defined in ISO/IEC 9075:1992.

**SQLCODE.** See ISO/IEC 9075:1992.

**Standard Map.** The Standard Map is a status map defined in SAMeDL_Standard that has the form "status Standard_Map as is_found uses boolean is (0 => true, 100 => false);". Standard_Map is the status map for fetch statements that appear in cursor declarations by default.

**Standard post processing.** The processing that occurs after the execution of an SQL procedure but before control is returned to the calling application.

**Static expression.** A value expression that can be evaluated at compile time (i.e., all the associated leaves consist solely of literals, constants, or domain parameter references).

**Status map.** A partial function that associates an enumeration literal or a raise statement with each specified list or range of SQLCODE values. Status maps are used within the abstract module to uniformly process the status data for all procedures.

**Target domain**. The domain of the object to which an assignment is being made in an assignment context.

**Universal constant.** A constant whose declaration does not contain a domain reference.

**Value expression.** A value expression differs from an SQL value expression in that (1) an operand may be a reference to a constant or a domain parameter, and (2) SAMeDL value expressions are strongly typed.

# Annex F: References

[1] Graham, Marc H.
*Guidelines for the Use of the SAME*.
Technical Report CMU/SEI-89-TR-16, ADA215846, Software Engineering Institute, May 1989.

[2] Chastek, Gary J., Graham, Marc H., Zelesnik, Gregory.
*Rationale for the SQL Ada Module Description Language, SAMeDL*.
Technical Report CMU/SEI-92-TR-16, Software Engineering Institute, 1992.

# Index

# Contents