Technical Report

CMU/SEI-90-TR-26
ESD-90-TR-227

The SQL Ada Module Description Language SAMeDL

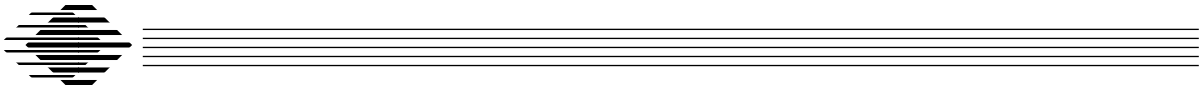
Gary J. Chastek
Marc H. Graham
Gregory Zelesnik

November 1990

Technical Report

CMU/SEI-90-TR-26
ESD-90-TR-227
November 1990

The SQL Ada Module Description Language
SAMeDL



Gary J. Chastek
Marc H. Graham
Gregory Zelesnik

Binding of Ada and SQL Project

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

This report was prepared for the SEI Joint Program Office HQ ESC/AXS

5 Eglin Street

Hanscom AFB, MA 01731-2116

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

FOR THE COMMANDER

(signature on file)

Thomas R. Miller, Lt Col, USAF, SEI Joint Program Office

This work is sponsored by the U.S. Department of Defense.

This work was created in the performance of Federal Government Contract Number F19628-95-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The Government of the United States has a royalty-free government-purpose license to use, duplicate, or disclose the work, in whole or in part and in any manner, and to have or permit others to do so, for government purposes pursuant to the copyright license under the clause at 52.227-7013.

This document is available through Asset Source for Software Engineering Technology (ASSET) / 1350 Earl L. Core Road ; P.O. Box 3305 / Morgantown, West Virginia 26505 / Phone: (304) 284-9000 / Fax: (304) 284-9001 / e-mail: sei@asset.com / WWW: http://www.asset.com/sei.html

Copies of this document are available through the National Technical Information Service (NTIS). For information on ordering, please contact NTIS directly: National Technical Information Service / U.S. Department of Commerce / Springfield, VA 22161. Phone: (703) 487-4600.

This document is also available through the Defense Technical Information Center (DTIC). DTIC provides acess to and transfer of scientific and technical information for DoD personnel, DoD contractors and potential con tractors, and other U.S. Government agency personnel and their contractors. To obtain a copy, please contact DTIC directly: Defense Technical Information Center / 8725 John J. Kingman Road / Suite 0944 / Ft. Belvoir, VA 22060-6218. Phone: 1-800-225-3842 or 703-767-8222.

Use of any trademarks in this report is not intended in any way to infringe on the rights of the trademark holder.

The SQL Ada Module Description Language SAMeDL

Abstract: This document is a reference manual for the SQL Ada Module Description Language (SAMeDL). The SAMeDL is used to describe database services needed by Ada application programs.

1. Introduction

The Structured Query Language (SQL) Ada Module Description Language (SAMeDL) automates the construction of software designed to use the SQL Ada Module Extensions application architecture [4]. Such applications are, in particular, written in Ada and use database management systems (DBMS) whose data manipulation language is SQL. The meaning of a SAMeDL compilation unit is given by:

- An SQL module, conforming to the SQL standard [2], possibly with DBMS implementation-specific extensions.
- An Ada compilation unit, conforming to the Ada standard [1].
- Rules concerning the relationship between the SQL and Ada texts.

1.1. Design Goals

The goal of the SQL Ada Module Extensions (SAME) is to produce robust database application programs written in a style suitable to Ada. The goal of the SAMeDL is to provide a binding of Ada and SQL based upon the SAME.

A style suitable to Ada means that an interface generated by the SAMeDL views the interaction between the application program and the database as a design object, separating database knowledge from the application program, and thereby yielding the potential for increased specialization of software development. Further, Ada treats SQL in the same way that it treats other foreign languages: that is, complete modules, rather than language fragments, are imported. Finally, Ada source programs do not contain embedded SQL statements. Ada and SQL are vastly different languages: Ada is a third-generation procedural language, while SQL is a data manipulation language in which the desired results are described. Text containing both Ada and SQL is therefore confusing and difficult to maintain. Note that this separation allows for the use of Ada tools such as syntax directed editors and debuggers.

The robustness of the SAMeDL is seen in its error handling, its uniform handling of null values, and its use of strong typing. SQL signals exceptional conditions through a status field, whose values have implementation-defined meanings. SAMeDL provides a

mechanism for specifying which DBMS exceptional conditions are expected, and presenting these conditions to the application in an application-defined, not DBMS-defined, manner. Exceptional conditions not handled by this mechanism cause the raising of a SAMeDL defined exception; thus DBMS errors cannot be ignored by the application. SQL supports incomplete information through the use of null values, but Ada has no such concept. Through the use of Ada's data abstraction mechanism, the SAMeDL offers automatic handling of incomplete information, ensuring that null values are never treated as not null. The use of strong typing in the SAMeDL further reduces the possibility of errors via type derivation, subtyping, the flagging of inappropriate operand usage, and runtime range constraint checking.

1.2. Language Summary

1.2.1. Overview

The SAMeDL is designed to facilitate the construction of Ada database applications that conform to the SAME architecture as described in [4]. The SAME extends the module language defined in the American National Standards Institute (ANSI) SQL standard to better fit the needs of Ada. 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 lower level 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.

1.2.2. Compilation Units and the Environment

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.

An environment exists that records information about the database (i.e., about schemas, tables, columns, views, and their associated domains), as well as information about modules and their content. A module must explicitly name the modules that it references.

1.2.3. 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 schemas, tables, views, and privileges. An abstract module generates 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.

1.2.4. Procedures and Cursors

A procedure declaration defines a basic database operation. The declaration generates 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 value, update statement, or select statement. The semantics of a SAMeDL statement directly parallel that of its corresponding SQL statement.

SAMeDL cursor declarations directly parallel SQL cursor declarations. They may contain an optional input parameter list, a query, an optional SQL "order by" clause, and cursor procedures. A cursor procedure contains an optional input parameter list, a cursor statement, and an optional status clause. A cursor statement may be an open statement, fetch statement, close statement, cursor update statement, or cursor delete statement. Only cursor statements may appear in a cursor procedure.

1.2.5. 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 specification and a body. The specification part of a base domain specifies the parameters needed to declare a domain derived from that base domain. A base domain body consists of patterns and options. Patterns contain a template 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.

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.

1.2.6. 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 generates an Ada exception declaration with the same name.

1.2.7. 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 those of Ada, in which a SAMeDL domain acts as an Ada type. As a consequence, domain conversions are introduced. These rules impact upon search conditions and select parameters.

1.2.8. 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 SQL_Database_Error. This prevents a database error from being ignored by the application.

1.2.9. Extensions

The SAMeDL may be extended without modification to the compiler 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.

1.2.10. 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.

1.3. Structure of the Standard

This reference manual contains five chapters. Each subsequent chapter is divided into sections that have a common structure. Each section introduces its subject, gives any necessary context-free syntax rules, and describes any further, context-sensitive rules that may apply. The section then gives the semantics of the construct just described syntactically.

In Chapters 3 and 4, semantics are given in part as the effect on an underlying ‘‘environment,’’ as that term is understood by SQL [2], Sections 4.6 and 4.7. In Chapter 5, semantics are defined in terms of the direct or indirect effects of the construct in the abstract interface (under the heading Ada Semantics) the concrete or SQL module (under the heading SQL Semantics), or the abstract module (under the heading Interface Semantics).

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".

1.4. Syntax and Other Notation

The context-free syntax of the language is described using a simple variant of the variant of BNF used in the *Reference Manual for the Ada Programming Language* ([1], Section 1.5). The variation is:

- Underscores are preserved when using the name of a syntactic category outside of a syntax rule ([1], Section 1.5(6)).

- 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 the Ada standard*, ([1], Section 2.3); whereas the category *SQL*_identifier is identical to the category identifier *as described in the SQL standard*, ([2], Section 5.3).

- 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.

The semantics of linguistic constructs are given in part by collections of string transformers (see Appendix C) that produce Ada and SQL texts from SAMeDL input. 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 (Section 5.6) is given by:

*Ada*_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.

2. 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.

2.1. 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 Section 2.4). 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

2.2. 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 sections. The remainder of this section 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 must 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.

2.3. Identifiers

Identifiers are, in general, Ada identifiers (see [1], Section 2.3). The exception is the use of *SQL*_identifiers (see [2], Section 5.3) as the names of **schemas**, **tables**, **views**, and **columns**. The major difference between *SQL*_identifiers and *Ada*_identifiers is that *SQL*_identifiers are limited to 18 characters in length, whereas *Ada*_identifiers are essentially unlimited in length. Note that an SQL reserved word may not appear at a point where the grammar specifies an *SQL*_identifier, and an Ada reserved word may not appear at a point where the grammar specifies an *Ada*_identifier.

The function SQL_{NAME} that appears in this language is an approximation of an injection (one-to-one function) from the set of *Ada*_identifiers to the set of *SQL*_identifiers. It is only an approximation, since no such injection exists.

2.4. Literals and Data Classes

Literals follow the SQL literal syntax ([2], Section 5.2). 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 | 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 ::= Ada_identifier
```

1. Each literal has an associated data class, denoted DATACLASS(*L)*, where *L* is a literal, which is determined in a straightforward way. 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 Subsection 4.1.1).

2. Every character_literal cl has an associated length LENGTH(cl) in the sense of [2], Section 5.2, rule 2.

3. Integer, fixed and float literals are collectively known as numeric literals. Every numeric literal *L* has a scale, SCALE(*L*). 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.

4. See Section 3.4 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.

2.5. 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.

2.6. Reserved Words

The following is the list of the SAMeDL keywords:

```
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
```

Of these keywords, the following must be reserved words:

```
ALL         ANY      AS           AVG       CHECK       COUNT     DEFAULT
DISTINCT    END      ENUMERATION  EXISTS    FROM        INTO      IS
KEY         MAP      MAX          MIN       NOT         NULL      PUBLIC
REFERENCES  SCALE    SELECT       SOME      STATUS      SUM       USER
VALUES
```

3. Common Elements

3.1. 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
```

The successful compilation of a module makes its name known to the environment.

SQL Semantics

There is an SQL module associated with each compilation unit and that contains the SQL semantics for that compilation unit. The name of the associated SQL module is implementation defined.

3.2. Context Clause

The context clause is a means by which a module gains visibility to names defined in other modules and in database schemas. The syntax and semantics of context clauses are similar to the syntax and semantics of Ada context clauses ([1], Section 8.4, Subsection 10.1.1) 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 ::=
    Ada_identifier

schema_name ::=
    SQL_schema_authorization_identifier

as_phrase ::=
    as Ada_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 must be the name of a definitional module and S the name of a schema known to the environment. 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 Sections 4.1, 4.2, and 5.1) is its exposed name within the text of that module. Within the context of any module, no two exposed names may be the same.

2. A module_name in a use_clause must 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 Sections 4.1, 4.2, and 5.1) 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.

3.3. Table Names and From Clause

The table names in insert, update, and delete statements and the from clauses of select statements, cursor declarations, and subqueries (see Sections 5.3, 5.4 and 5.12) also make names, in particular column names, visible within an abstract module. The from_clause differs from an *SQL*_from_clause ([2], Section 5.20) 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_name.] SQL_identifier
```

1. If present, schema_name must be the name of a schema known to the environment and that is either the schema named in the authorization clause or in the context of the abstract module in which table_name appears. If the schema_name is absent from the table_name, then the *SQL*_identifier must be the name of a table within the schema 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 in the table_ref is *exposed*; otherwise the table_name is hidden and the correlation_name is exposed. No two exposed names within a from_clause may be the same.

3. For the scope of table_names see [2], Section 5.20, syntax rule 4; Section 8.5, rule 3; and Section 8.12, rule 5.

3.4. 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 section, an *item* is either:

- A definitional module (Section 4.1), an abstract module (Section 5.1), or a schema module (Section 4.2).
- A procedure (Section 5.2), a cursor (Section 5.4), or a procedure within a cursor (Section 5.5).
- Anything in the syntactic category 'definition' as given by Section 4.1. This includes base domains, domains, subdomains, domain parameters, enumerations, constants, records, and status maps.
- An enumeration literal within an enumeration (Section 2.4 and Subsection 4.1.7).
- An exception (Subsection 4.1.8).
- An input_parameter of a procedure or cursor declaration (Sections 5.2, 5.4, and 5.6).
- A database table defined within a schema module (Section 4.2).
- A database column defined within a database table (Section 4.2).

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 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 section. When these rules determine more than one meaning for an identifier, then all items referenced shall be enumeration literals.

```
module_reference ::= Ada_identifier

schema_reference ::= schema_name | Ada_identifier

base_domain_reference ::= [module_reference.]Ada_identifier

domain_reference ::= [module_reference.]Ada_identifier

domain_parameter_reference ::= domain_reference.Ada_identifier

subdomain_reference ::= [module_reference.]Ada_identifier

enumeration_reference ::= [module_reference.]Ada_identifier

enumeration_literal_reference ::= [module_reference.]Ada_identifier

exception_reference ::= [module_reference.]Ada_identifier

constant_reference ::= [module_reference.]Ada_identifier

record_reference ::= [module_reference.]Ada_identifier

procedure_reference ::= [module_reference.]Ada_identifier

cursor_reference ::= [module_reference.]Ada_identifier

cursor_proc_reference ::= [cursor_reference.]Ada_identifier

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

status_reference ::= [module_reference.]Ada_identifier

table_reference ::= [schema_reference.]SQL_identifier
        (See [2], 5.4 and 5.20)

column_specification ::= [table_reference.]SQL_identifier
        (See [2], 5.7)
```

A reference is a simple name (an identifier) optionally preceded by a prefix: a sequence of as many as three identifiers, separated by dots. Unlike Ada ([1], Sections 8.2 and 8.3), it is necessary to treat the prefix as a whole, not a component.

For the purposes of this section, 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.

Prefix Denotations

The prefix of a reference may denote any one of the following:

- A procedure, cursor or cursor procedure, but only from within the text of the procedure, cursor, or cursor procedure.

- A database table, if the table is in scope at the location in which the reference appears.

- A domain.
- A definitional or schema module.

Note: A prefix must have exactly one denotation.

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 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 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 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* (Section 3.3, rule 3), and either
      i. *P* 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 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 database 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 database 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 Sections 5.4, 5.2, and 5.5) 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*.

Examples

The following section 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;
```

3.5. Value Expressions and Assignment Contexts

A value expression (see Section 5.10) is said to appear in an *assignment context* if it is either

- The static expression in a constant declaration (see Subsection 4.1.5),
- A select parameter (see Section 5.7),
- A value in an insert_values_list (see Section 5.8), or
- The right hand side of a set_item within an update_statement (see Section 5.3).

Such a value expression *VE* has two associated domains: the target domain, which is inferred by the value expression's context, and its domain, DOMAIN(*VE*), which is determinable without reference to the context (see Section 5.10). A value expression in an assignment context is said to *conform* to its target domain if

- If DOMAIN(*VE*) $\neq$ NO_DOMAIN, then DOMAIN(*VE*) is the target domain.
- If the data class of the target domain is **integer** or **fixed**, then the data class of DOMAIN(*VE*) is **integer** or **fixed**.
- If the data class of the target domain is **float**, then the data class of DOMAIN(*VE*) is **integer**, **fixed**, or **float**.
- If the data class of the target domain is **character**, then the data class of DOMAIN(*VE*) is **character**, and LENGTH(*VE*) $\leq$ the maximum length associated with the target domain.

- If the data class of the target domain is **enumeration**, then if DOMAIN(*VE*) = NO_DOMAIN, then *VE* is an enumeration literal in the target domain.

3.6. 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. If a status map is attached to the procedure (see Section 5.13), then if that map contains an sqlcode_assignment whose left-hand side is equal to the value of the SQLCODE parameter, then the Ada procedure's status parameter is set to the value of the right hand side of that sqlcode_assignment, if that right hand side is an *Ada*_enumeration_literal: if that right hand side is a **raise** statement, then the named *Ada*_exception is raised. Note that 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.

2. If the value of the SQLCODE parameter is not matched by the left hand side of any sqlcode_assignment in the map attached to the procedure *or* there is no status map attached to the procedure and the value of the SQLCODE parameter is other than zero, then an error condition exists. In this case the parameterless procedure SQL_Database_Error_Pkg.Process_Database_Error is called. The exception SAMeDL_Standard.SQL_Database_Error is raised.

3.7. 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 may be expressed 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 Sections 3.5 and 5.10.
- Search conditions by adding atomic predicates. Operands of those atomic predicates shall likewise be restricted according to rules such as those in Section 5.11.
- Input parameter lists by adding the mode **out** to the parameter declarations, according to the rules of Section 5.6.
- Table elements. If the extended table element is in the form of a column definition, the domain reference must be present (see Subsection 4.2.1).

Similarly, database data returned from extended procedures and cursors shall be defined in the syntax and semantics of select parameter lists (Section 5.7) with the syntax and semantics of the extended value expression class replacing the standard syntax and semantics. Such outputs are record objects. An extended statement returning such data will contain an into_clause as described in Section 5.9 for specifying the record parameter name and type.

4. Data Description Language and Data Semantics

4.1. 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]
    definition module Ada_identifier_1 is
        {definition}
    end [Ada_identifier_2];

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

When present, *Ada*_identifier_2 must equal *Ada*_identifier_1.

The successful compilation of a definitional module makes its name, *Ada*_identifier_1, known to the environment.

No with_schema_clause may appear in the context of a definitional module.

No two declarations within a definitional module shall have the same name, except for enumeration literals (see Subsection 4.1.7).

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, *Ada*_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.

4.1.1. Base Domain Declarations - Specifications

A base domain declaration has both a specification and a body. The specification specifies what must be known to declare a domain from the base domain.

```
base_domain_declaration ::=
    base domain Ada_identifier_1 is
        {base_domain_parameter}
    end [Ada_identifier_2];

base_domain_parameter ::=
    Ada_identifier : data_class [ := static_expression] ; |
    map := pos | image ;
```

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

1. If present, *Ada*_identifier_2 must equal *Ada*_identifier_1. *Ada*_identifier_1 is the name of the base domain.

2. 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 Subsection 4.1.4). The static_expression within a base_domain_parameter, when present, specifies a default value for the parameter. This default value must be of the correct data_class, that is, in the parameter declaration

   ```
   Id : class := expr;
   ```

   DATACLASS(expr) must be the value of class.

3. A domain is classified by its data_class. That is, an enumeration domain is a domain whose data_class is enumeration, a fixed domain is a domain whose data_class is fixed, etc.

4. 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 Subsection 4.1.4) are not of any of the data classes listed above. The value of an **enumeration** parameter is an enumeration_reference (see Section 3.4); the value of **map** is a database_mapping (see Subsection 4.1.4). A base domain declaration may explicitly declare a **map** parameter for the purpose of assigning a default mapping.

   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, will be used to translate enumeration literals to their database encodings. This is similar for **image** and the 'IMAGE attribute. See [1], annex A, Subsection 4.1.4 and Section 4.3.

5. Every fixed domain has a predefined parameter **scale** whose value is implementation defined ([1], Section 5.5).

Examples

The SAME standard domain SQL_Int is defined by

```
base domain SQL_Int is
    First : integer;
    Last  : integer;
end SQL_Int;
```

The SAME standard domain SQL_Char is defined by

```
base domain SQL_Char is
    Length : integer;
end SQL_Char;
```

The SAME standard domain SQL_Enumeration_as_Integer is defined by

```
base domain SQL_Enumeration_as_Integer is
    map := pos;
end SQL_Enumeration_as_Integer;
```

4.1.2. Base Domain Declarations - Bodies

Base domain bodies are used to describe the Ada declarations to be used in declaring the Ada constructs that implement the domain (see Subsection 4.1.4). They are also used to give values to concepts (called options, see below) that are essential to the definition of domains based on the base domain.

```
base_domain_body_declaration ::=
    base domain body Ada_identifier_1 is
        patterns
        options
    end [Ada_identifier_2];
```

When present, *Ada*_identifier_2 must equal *Ada*_identifier_1.

Patterns

The patterns portion of a base domain body 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

subdomain_pattern ::=
    subdomain pattern is pattern_list

derived_domain_pattern ::=
    derived domain pattern is pattern_list

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 Subsection 4.1.4). 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 Subsection 4.1.4.

For a parameter to be recognized as such in a pattern, it must be 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 name of the domain or subdomain being declared. 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 the name of parent domain (see Subsection 4.1.4).

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 Subsection 4.1.4); 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.

Example

The domain pattern for the SAME standard base domain SQL_Int is given by:

```
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);'
```

The parameters First and Last are used to define an integer range constraint. That constraint will appear in the instantiated pattern only if both First and Last are assigned values in the domain declaration. See Subsection 4.1.4.

The domain pattern for the SAME standard base domain SQL_Char is given by:

```
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 ([Length]);'
    'package [self]_Ops is new SQL_Char_Ops ('
        '[self]_Base, [self]NN_Base);'
```

Options

```
options ::=
    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 ; |
    for word_list use pattern_list ;

dbms_type ::= int | integer | smallint |
    dec | decimal | real | float |
    double precision | numeric |
    char | character

type ::= dbms | not null | null

converter ::=
    function {pattern_list}; |
```

procedure {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. Implementations are free to add options beyond those given above. The meanings of the predefined 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 Sections 5.6 and 5.7.

2. The **data class** option is the data class (see Section 2.4) 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 Subsection 4.1.4) on *BD*, then DATACLASS(D)=*dc*. The data class governs the use of literals with such objects (see Sections 5.8 and 5.10).

3. The **dbms type** of a base domain is the *SQL*_data_type ([2], Section 5.5) to be used at the level of the interface for all objects of domains based directly or indirectly on the base domain. See Sections 5.6, 5.7, and 5.8.

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 Subsection 4.1.4) and the database type associated with a domain. A method may be a function, a procedure, 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, 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 Subsections 4.1.1 and 4.1.4.

Example

Here are options describing the SAME standard domain SQL_Int.

```
for data class use integer;
for dbms type use integer;
for not null type name use '[self]_not_null';
for null type name use '[self]_type';
for conversion from dbms to not null use type mark;
for conversion from not null to dbms 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';
```

The conversions from and to the not null type interoperate with the database type

(SQL_Standard.Integer). They are Ada type conversions. The conversions from and to the null-bearing type interoperate with the not null type. Those functions are found in a package instantiated for each domain. Notice the use of the domain parameter in those options. Notice also that an assign procedure is identified, as the null-bearing type is limited.

The SAME standard base domain SQL_Char can be described by the following options:

```
for data class use character;
for dbms type use character '([length])';
for not null type name use '[self]_not_null';
for null type name use '[self]_type';
for conversion from dbms to not null use type mark;
for conversion from not null to dbms 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';
```

Notice that the database type is parameterized by the parameter length. Again, the null-bearing type is limited and an assignment procedure for it must be given. In this case, the procedure is declared in the Ada package with the types.

4.1.3. The SAME Standard Base Domains

The SAME standard base domains are assumed to be defined in every environment in a predefined definitional module named SAMeDL_Standard. They are: SQL_Int, SQL_Smallint, SQL_Char, SQL_Real, SQL_Double_Precision, SQL_Enumeration_as_Char, and SQL_Enumeration_as_Int. The declarations assumed for these base domains appears in Appendix C.

4.1.4. Domain and Subdomain Declarations

```
domain_declaration ::=
    domain Ada_identifier is new bas_dom_ref [not_null]
        [ ( parameter_association_list ) ] ;

subdomain_declaration ::=
    subdomain Ada_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 ::=
    base_domain_parameter => static_expression |
    map => database_mapping |
    enumeration => enumeration_reference
    scale => 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 Subsection 4.1.2.

4. The data class of a domain *D*, denoted DATACLASS(*D*), is the data class of its base domain, the value of **for data class** option. A domain is numeric if its data class is numeric (see Section 2.4). Every numeric domain has a scale. Integer domains have scale 0. All float domains have the same scale, which is greater than the scale of any non-float domain or object. All fixed domains have a scale which is supplied as the value of the **scale** parameter and that must appear in any declaration of a fixed domain. The values that may be assigned to the **scale** parameter in the declaration of a fixed domain are implementation defined.

5. Every parameter, except for **scale**, **enumeration** and **map**, in a parameter_association_list must be a parameter of the declaration of the referenced base domain. See Subsection 4.1.1.

6. All base domain parameters which appears in a non-optional pattern or option phrase in a domain's base domain must have an assigned value (see Subsection 4.1.2); otherwise, the domain declaration is in error.

7. Suppose *param* is a parameter of domain or subdomain *D*, and

   ```
   param : C := E
   ```

   appears in the base domain of *D*. Then DATACLASS(*param*) is C.

8. A domain or subdomain declaration *D* is said to assign the value *v* to the parameter *param*, if

a. the parameter_association *param* => *E* appears in *D* and *VALUE(E)*=*v*; or

b. (a) does not hold, *D* is a subdomain_declaration or the declaration of a derived domain, and the parent domain assigns the value *v* to the parameter *param*; or

c. (a) and (b) do not hold and in the base_domain_declaration for the base domain of *D*, the base_domain_parameter

```
param : class := E
```

appears and the static value VALUE(E) = *v*

In all cases, the data class of *v* shall be the data class assigned to *param* in the base_domain_declaration.

9. An enumeration domain or subdomain declaration *D* is said to assign the value *v* to the enumeration literal *El*, if

   a. The parameter_association "**map** => database_mapping" appears in *D* and that database_mapping

      - is **pos**, and v = **pos**(*El*), or
      - is **image**, and v = **image**(*El*), or
      - contains an enumeration_association of the form

        El => *v*

      or

   b. (a) does not hold, *D* is a subdomain_declaration or the declaration of a derived domain and the parent domain assigns the value *v* to the enumeration literal *El*; or

   c. (a) and (b) do not hold and in the base_domain_declaration for the base domain of *D*, either

      - the base_domain_parameter

        map := **pos**

        appears and **pos**(*El*) = *v*, or
      - the base_domain_parameter

        map := **image**

        appears and **image**(*El*) = *v*. See Subsection 4.1.1 for the meaning of the database_mappings **pos** and **image**.

10. The parameters **enumeration** and **map** may appear only if the domain being declared is an enumeration domain; in that case, the **enumeration** parameter must appear. If an enumeration_association_list is present as a database mapping, then it must satisfy the following:

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.

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 Subsection 4.1.2 and Section 4.3.

11. 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 having 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 Subsection 4.1.2, shall appear within the Ada package specification corresponding to the definitional 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.

Examples

The following examples illustrate the declaration of domain objects, and have been annotated with references to the appropriate sections of the language definition. The base domains used in these examples exist in the predefined definitional module SAMeDL_Standard. The constant Max_SQL_Int is declared in the predefined definitional module SAMeDL_System. Both SAMeDL_Standard and SAMeDL_System are assumed to be visible, as is the enumeration declaration *Colors* (see Subsection 4.1.7).

```
domain Weight is new SQL_Int ( -- 4.1.4: #1a
  First => 0,                  -- 4.1.4: #5 and #8
  Last  => Max_SQL_Int);       -- 4.1.4: #5 and #8

domain City is new SQL_Char (  -- 4.1.4: #1a
  Length => 15);               -- 4.1.4: #5 and #7

domain Color is new SQL_Enumeration_As_Char ( -- 4.1.4: #1a
  enumeration => Colors,                       -- 4.1.4: #10
  map         => image);                       -- 4.1.4: #9

domain Auto_Weight is new Weight ( -- 4.1.4: #1b
  Last => 10000);                  -- 4.1.4: #5 and #8

subdomain Auto_Part_Weight is Auto_Weight ( -- 4.1.4: #2
  Last => 2000); -- 4.1.4: #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 Ada code below is the instantiation of the domain pattern
--   from base domain SQL_Char

type CityNN_Base is new SQL_Char_Not_Null;
subtype City_Not_Null is CityNN_Base (1 .. 15);
type City_Base is new SQL_Char;
subtype City_Type is City_Base (City_Not_Null'Length);
package City_Ops is new SQL_Char_Ops(City_Type, City_Not_Null);

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

package Color_Pkg is new SQL_Enumeration_Pkg(Colors);
type Color_Type is new Color_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);
```

4.1.5. Constant Declarations

```
constant_declaration ::=
    constant Ada_identifier [: domain_reference [not_null] ]
        is static_expression ;

static_expression ::=
    value_expression
```

A static expression is a value expression (see Section 5.10) 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 [not null]] is E ;
```

1. DATACLASS(*K*) is DATACLASS(*E*), the datatype of the expression E. See Sections 2.4 and 5.10.

2. If DATACLASS(*K*) is **enumeration**, then *D* must be present and must name an enumeration domain of which *E* is an enumeration literal.

3. If DATACLASS(*K*) is **character**, then *D* must be present and must name a character domain.

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*).

5. If the domain_reference *D* is present, then

   a. *D* is the *target domain* of the static expression *E*.

   b. DOMAIN(*K*)=*D provided E* conforms to *D* (see Section 3.5).

   c. If DATACLASS(*K*) is numeric, then SCALE(*K*) = SCALE(*D*), *provided* SCALE(*E*) ≤ SCALE(*D*).

   d. AdaTYPE(*K*) is defined as follows:

      i. If either **not null** appears in *K* or else *D* is defined as not null only, then AdaTYPE(*K*) is the type name within *D* designated as not null bearing;

      ii. otherwise, AdaTYPE(*K*) is the null-bearing type name within *D*.

Ada Semantics

Let VALUE represent the function which calculates the value of a static_expression using the rules of SQL. 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 [`**not null**`]]` **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).

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

Again, let *K* denote the constant declaration

constant `C [: D [`**not null**`]]` **is** `E ;`

Let *Q* be the Ada representation of VALUE(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

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

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

SQL Semantics

See section 4.3.

4.1.6. Record Declarations

```
record_declaration ::=
    record Ada_identifier_1 [named_phrase] is
        component_declarations
    end [Ada_identifier_2] ;

named_phrase ::=
    named Ada_identifier

component_declarations ::=
    component_declaration {component_declaration}

component_declaration ::=
    component_name {, component_name} : domain_reference [not_null] ;

component_name ::= Ada_identifier
```

If present, *Ada*_identifier_2 must be equal to *Ada*_identifier_1. Let *R* be a record declaration. Define the name 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 Sections 5.2, 5.5, and 5.9.

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 *Ada*_identifier_1.

2. R_{Ada} shall be equivalent, in the sense of [1], Subsections 3.2.10 and 3.7.2, to the record type R_1, which is defined as follows. If R contains *n* component declarations, and the i^{th} component declaration of *R* is given by

   $C_1, \ldots, C_m$: D [**not null**]

   then the i^{th} component declaration of R_1 shall be

   $C_1, \ldots, C_m$: T

   where T is an Ada type name determined to be:

   a. The not null-bearing type name within the domain D, if either D is a not null only domain or **not null** is present in the i^{th} component of *R*;

   b. Otherwise the null-bearing type name within the domain D.

Examples

The following example illustrates the declaration of a record object.

```
record Parts_Row_Record_Type named Parts_Row_Record is
  Part_Number      : Pno not null;
  Part_Name        : Pname;
  Color            : Color;
  Weight_In_Ounces : Weight;
  City             : City;
end Parts_Row_Record_Type;
```

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

```
type Parts_Row_Record_Type is record   -- 4.1.6: Ada Semantics #1
  Part_Number      : Pno_Not_Null;  -- 4.1.6: Ada Semantics #2
  Part_Name        : Pname_Type;
  Color            : Color_Type;
  Weight_In_Ounces : Weight_Type;
  City             : City_Type;
end record;
```

4.1.7. Enumeration Declarations

Enumerations are used to declare sets of enumeration literals for use in enumeration domains and status maps. See Subsections 4.1.4 and 4.1.9.

```
enumeration_declaration ::=
    enumeration Ada_identifier_1 is ( enumeration_literal_list ) ;

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

```
enumeration_literal ::=
    Ada_identifier
```

1. Ada_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 Ada_identifier_1 is ( enumeration_literal_list ) ;
```

Note: Ada character literals may not be used in enumerations.

Examples

The following are examples of enumeration declarations. They are assumed to be defined in some definitional module.

```
enumeration Operation_Status is (Disk_Error,
  Data_Conversion_Error, Invalid_SQL_Statement, Not_Found);

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

The above declarations produce the following Ada code.

```
type Operation_Status is (Disk_Error,
  Data_Conversion_Error, Invalid_SQL_Statement, Not_Found);

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

4.1.8. Exception Declarations

```
exception_declaration ::=
    exception Ada_identifier ;
```

Ada_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

```
Ada_identifier_1 : exception ;
```

Examples

The following are examples of exception declarations. They are assumed to be defined in some definitional module.

```
exception Data_Definition_Does_Not_Exist;

exception Insufficient_Privilege;
```

The above declarations produce the following Ada code.

```
Data_Definition_Does_Not_Exist : exception;

Insufficient_Privilege : exception;
```

4.1.9. Status Declarations

The execution of any procedure (see Sections 3.7, 5.2, and 5.5) causes the execution of an SQL procedure. That execution causes a special parameter, called the SQLCODE 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" [2], Subsection 4.10.1. Status maps are used within abstract modules to process the status data in a uniform way for all procedures. Each map declares a partial function from the set of all possible SQLCODE values onto (1) elements of a SAME enumeration and (2) raise statements. **Note**: The function is DBMS specific in that SQLCODE values are not specified by standard SQL [2], whereas the enumeration type and exceptions are not specific to any DBMS.

Status Map Declarations

```
status_map_declaration ::=
    status Ada_identifier_1
        [named_phrase]
        [uses target_enumeration]
    is ( sqlcode_assignment {, sqlcode_assignment} );

target_enumeration ::=
    enumeration_reference | boolean

sqlcode_assignment ::=
    static_expression_list => Ada_enumeration_literal |
    static_expression_list => raise exception_reference

static_expression_list ::=
    static_expression { , static_expression } |
    static_expression .. static_expression
(see [1] 3.5)
```

1. *Ada*_identifier_1 is called 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 sqlcode_assignments that contain **raise** may be present in the status_map_declaration.

4. Every *Ada*_enumeration_literal within an sqlcode_assignment must be an *Ada*_enumeration_literal within the enumeration referenced by the target_enumeration.

5. If E => L (or E => **raise** X) is an sqlcode assignment then

   - DATACLASS(E) = **integer**;

   - If E' => L' (or E' => **raise** X') is any other sqlcode_assignment within the status_map_declaration, then E and E' shall not evaluate to the same integer.

   Note: An sqlcode_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 sqlcode_assignments, however.

6. Every environment shall be assumed to contain the definition of a status map Standard_Map, defined as follows:

```
status Standard_Map named Is_Found  uses boolean is
    (0 => True, 100 => False);
```

   Note: Standard_Map is the status map for those fetch statements that appear in cursor declarations by default. (See Section 5.5.) It signals end of table by returning **false**.

Examples

The following is an example of a status map declaration. It is assumed to be defined in some definitional module. For the enumeration and exception declarations, refer to the examples in Subsections 4.1.7 and 4.1.8. They are assumed to be visible at the point at which the status map is declared.

```
status Operation_Map named Result_of_Operation
  uses Operation_Status is (

    -600 .. -699      => Disk_Error,
    -500 .. -599      => Data_Conversion_Error,
    -300 .. -499      => Invalid_SQL_Statement,
    -101, -110, -113  => raise Data_Definition_Does_Not_Exist,
    -25               => raise Insufficient_Privilege,
     100              => Not_Found);
```

A status map declaration is used to specify the standard post-processing following the execution of an SQL procedure as described in Section 3.6.

4.2. Schema Modules

```
schema_module ::=
    [context]
    [extended] schema module SQL_identifier_1 is
        {schema_element}
    end [SQL_identifier_2];

schema_element ::=
```

```
    table_definition | view_definition | SQL_privilege_definition |
    extended_schema_element

SQL_privilege_definition ::=
     (see [2] 6.10)
```

1. If present, *SQL*_identifier_2 must equal *SQL*_identifier_1. *SQL*_identifier_1 is the *name* of the schema_module.

2. *SQL*_identifier_1 must be different from any other schema name known to the environment ([2], Section 6.1).

3. A table_definition or view_definition may be extended only if the keyword **extended** appears in the associated schema module declaration.

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

SQL Semantics

An SQL schema is formed from a schema_module by

- dropping the optional context, if present
- dropping **extended**, if present
- replacing the "**schema module**" string by "create schema authorization"
- dropping the string "**is**"
- dropping the trailing "**end** [*SQL*_identifier_2] ;"

where the schema_elements are replaced as specified in Subsections 4.2.1 and 4.2.2.

4.2.1. Table Definitions

```
table_definition ::=
    [extended]table SQL_identifier_1 is
        table_element {, table_element}
    end [SQL_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 [2] 6.4)
```

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

SQL_unique_specification ::=
    (see [2] 6.6)

SQL_reference_specification ::=
    (see [2] 6.7)

table_constraint_definition ::=
    SQL_unique_constraint_definition |
    SQL_referential_constraint_definition |
    check_constraint_definition

SQL_unique_constraint_definition ::=
    (see [2] 6.6)

SQL_referential_constraint_definition ::=
    (see [2] 6.7)

check_constraint_definition ::=
    check ( search_condition )
```

1. If present, *SQL*_identifier_2 must equal *SQL*_identifier_1. *SQL*_identifier_1 is the *name* of the table and the table_definition.

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

3. A table_definition must 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 [2], Sections 6.3 and 6.6; for the semantics of **check**, see [2], Sections 6.3 and 6.8.

7. If **extended** appears in a table_definition then **extended** must also appear in the associated schema_module declaration.

8. If an extended_table_element appears in a table_definition, then the keyword **extended** must appear in that table_definition (see Section 3.7).

SQL Semantics

An SQL table definition is formed from a table definition by prepending the string "create" and dropping

- **extended**, if present
- the string "**is**"
- the final "**end** [*SQL*_identifier_2] ;"
- the ":" and the domain_reference from each column_definition, and replacing it with the DBMS_TYPE of the specified domain (see section 4.1.2) if the *SQL*_data_type is not specified
- the trailing ";" from each column_definition

If domain *D* is assigned to a column *C*, then conversion between DBMS_TYPE(*D*) and DBMS_TYPE(*C*) shall be legal in both directions by the rules of SQL ([2], Section 8.6, syntax rule 3; Section 8.7, syntax rule 6; etc).

4.2.2. View Definitions

```
view_definition ::=
    [extended] view SQL_identifier_1 as query_spec
        [with check option]
    end [SQL_identifier_2];

query_spec ::=
    query_specification |
    extended_query_specification
```

1. If present, *SQL*_identifier_2 must equal *SQL*_identifier_1. *SQL*_identifier_1 is the *name* of the view and the view_definition.
2. The name of the view_definition must 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.

SQL Semantics

An SQL view definition is formed from a view_definition by

- prepending the string "create"
- dropping **extended**, if present
- adding an SQL view column list enclosed in parenthesis. The view column list is composed from the select parameter list of the associated query_spec (see Section 5.7);

that is, let {SP_1 ... SP_n} be the select parameters in the query_specification. Then the generated SQL view column list is

AdaNAME(SP_1) , ... , AdaNAME(SP_n)

with the restriction that if i ≠ j, then AdaNAME(SP_i) ≠ AdaNAME(SP_j).

- transforming the query_specification to an SQL query specification as described in Section 5.7, with the added restrictions mentioned in the previous item

- dropping the final "**end** [*SQL*_identifier_2] ;"

If the query_spec is an extended_query_specification, see Section 3.7.

4.2.3. Examples

The following is an example of a schema module.

```
schema module Parts_Suppliers_Database is

  -- the Parts table

  table P is                                  -- 4.2.1: #1 and #2

    PNO not null : Pno,                       -- 4.2.1: #3 and #4
    PNAME        : Pname,                     -- 4.2.1: #5
    COLOR        : Color,
    WEIGHT       : Weight,
    CITY         : City,
    unique (PNO)

  end P;

  -- the Suppliers table

  table S is                                  -- 4.2.1: # 1 and #2

    SNO not null : Sno,                       -- 4.2.1: #3 and #4
    SNAME        : Sname,                     -- 4.2.1: #5
    SSTATUS      : Sstatus,                   -- "Status" is a reserved word
    CITY         : City,
    unique (SNO)

  end S;

  -- the Orders table

  table SP is                                    -- 4.2.1: #1 and #2

    SNO character(5) not null : Sno,        -- 4.2.1: #3 and #4
    PNO character(6) not null : Pno,
    QTY integer               : Quantity, -- 4.2.1: #5
    unique (SNO, PNO)

  end SP;

  -- the Part_Number_City view

  view PNO_CITY as                            -- 5.2.2: #1 and #2
```

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

      end PNO_CITY;

    end Parts_Suppliers_Database;
```

4.3. Data Conversions

The procedures that are described in an abstract module (Chapter 5) 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 must be 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 section.

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 Subsection 4.1.2) is executed.

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 Sections 5.6 and 5.8.) 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 [2], Subsection 4.10.2 and Section 5.6, General Rule 1.) 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 (Subsection 4.1.2).

For *output parameters*, record components in **fetch** or **select** statements, this process is run in reverse. If the record component has a null-bearing type, then there is an associated *SQL*_parameter_specification to which an *SQL*_indicator_parameter has been assigned. (See Section 5.7.) If the indicator parameter is negative, the record component is set to the null value. Otherwise, the record parameter is set from the SQL parameter by the conversion process identified for the base domain. (*Note*: SQL requires that an indicator parameter be specified if a target parameter of a fetch or select statement is to be assigned a null value. See [2], Section 8.6, General Rule 8; Section 8.10, General Rule 8. In practice this means that, if a given output record component has a not null-bearing type, and a null value for that component appears at the SQL module, the behavior of the system is defined by the DBMS implementation.)

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 Section 5.8) and value expressions (see Section 5.10) 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 Subsections 4.1.2 and 4.1.4).

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.

If *V* is a reference to

- a constant declared by

  constant `C [: D [`**not null**`]]` **is** `E ;`

- a domain parameter *param* of domain D, and D assigns the expression E to *param* (see Subsection 4.1.4),

- or an enumeration literal *El* from enumeration domain D (see Sections 5.3, 5.8, 5.10, and 5.11), and D assigns the expression E to *V*,

then *V* is replaced by the static expression $SQL_{VE}(E)$.

5. Abstract Module Description Language

5.1. Abstract Modules

```
abstract_module ::=
    [context]
    [extended] abstract module Ada_identifier_1 is
        authorization schema_reference
        {definition}
        {procedure_or_cursor}
    end [Ada_identifier_2];

procedure_or_cursor ::=
    cursor_declaration | procedure_declaration
```

1. When present, *Ada*_identifier_2 must equal *Ada*_identifier_1. *Ada*_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 [2], Sections 6.1, 6.10, 7.1, 8.4, 8.5, 8.7, 8.10, 8.11, and 8.12.

Note: Unlike the description of a module in [2], Section 7.1, it is not necessary for all cursor declarations to precede all other statements within an abstract module.

A procedure_or_cursor may be an extended procedure or an extended cursor (see Sections 5.2 and 5.4) 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, *Ada*_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.

5.2. Procedures

The procedures discussed in this section are not associated with a cursor. Cursor procedures are discussed in Section 5.5.

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 Section 5.1) 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 Ada_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
```

*Ada*_identifier_1 is the name of the procedure.

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

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** must 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:

1. 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.

2. The simple name of P_{Ada} is 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 Section 5.6.)

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. Let IC be the into_clause appearing (possibly by assumption) 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 Section 5.9.

   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 Section 5.7 has been applied.) Then the row record type is equivalent in

the sense of [1], Subsections 3.2.10 and 3.7.2, to the record whose i^{th} component $COMP_{Ada}(SP_i)$ (for $i \leq m$) is given by

```
AdaNAME(SP_i) : AdaTYPE(SP_i)
```

provided that AdaNAME(SP_i) is not NO_NAME (see Section 5.7).

The mode of the row record parameter shall be **in out**.

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. Let *IC* be the insert_from_clause appearing (possibly by assumption) 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 Section 5.9.

   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 columns 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 [1], Subsections 3.2.10 and 3.7.2, to the record whose i^{th} component $COMP_{Ada}(C_i)$ for $1 \leq i \leq m$ is given by

   ```
   AdaNAME(C_i) : AdaTYPE(C_i)
   ```

   (See Section 5.8).

   The mode of the record parameter is **in**.

4. If the statement within the procedure is an extended_statement, see Section 3.7; for extended parameter lists, see Section 5.6.

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 Subsection 4.1.9 and Section 5.13.

SQL Semantics

Each procedure declaration P shall be assigned an SQL procedure P_{SQL} within the SQL module for the compilation unit. P_{SQL} has three parts:

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

2. A list of *SQL*_parameter_declarations. An SQLCODE parameter is declared for every SQL procedure. 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 Section 5.6, 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 columns in the insert_column_list whose corresponding entry in the insert_value_list is an *SQL*_column_name (thus not a literal or constant_reference). See Section 5.8.

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 Section 5.7.

d. If the statement is an extended_statement, see Section 3.7.

3. An *SQL*_SQL_statement ([2], Section 7.3). This is derived from the statement in the procedure declaration. See Sections 3.7 and 5.3.

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 Section 5.6) are set from the value of $PARM_{Ada}(INP)$ (see Ada Semantics above) according to the rule of Section 4.3. 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 Section 5.8.

2. Standard post processing, as described in Section 3.6, is performed.

3. If the value of the SQLCODE parameter is zero, and the statement within the procedure is a select_statement, then the value of the component of the row record parameter $COMP_{Ada}(SP_i)$ is set from the values of the actual parameters associated with the SQL formal parameters $SQL_{SPNAME}(SP_i)$ and $INDIC_{NAME}(SP_i)$ (see Section 5.7), according to the rule of Section 4.3.

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 : Pname) 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 Pname_Type;          -- 5.6: Ada Semantics #1
                                          -- 5.2: Ada Semantics
  Delete_Status : out Operation_Status); -- 5.2: Ada Semantics #5
```

5.3. Statements

This section 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_specification = update_value

update_value ::=
        null | value_expression

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

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

```
from Row : new Row_Type
```

See Section 5.9.

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

   ```
   into Row : new Row_Type
   ```

   See Section 5.9.

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 ([2], Subsection 8.7.3). (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 may cease functioning, where a program supplying an insert_column_list would continue to operate correctly.)

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 must conform, as described in Section 5.8.

5. If the statement is an insert_statement_query, then let C_1, C_2, . . . , C_m be the columns appearing in an insert_column_list, and for each $1 \le i \le m$, let D_i be the domain assigned to the column C_i in the environment (Section 4.2). The select_parameters in the select_list of the query_specification shall not specify a named_phrase or a not_null phrase. Let SP_1, SP_2, . . . , SP_n be the elements of that select_list. Then the select_list and the insert_column_list shall have the same length, that is m shall equal n. For each $1 \le i \le m$, D_i is the *target domain* of SP_i, and SP_i shall conform to D_i (see Section 3.5).

6. The following apply to update statements. Let

   ```
   C = v
   ```

   be a set_item within an update_statement. Let D be the domain assigned by the environment to the column C. Then

   a. If v is **null**, D shall not be defined as a not null-bearing domain.

   b. Otherwise, v is a value_expression. The *target domain* of v is D, and v shall conform to D (see Section 3.5).

SQL Semantics

This section describes the text of an SQL statement corresponding to a 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 Section 5.11 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_{SC} described in Section 5.11 to the search_conditions, if any, in the query_specification.

   b. Applying the transformation SQL_{VE} defined in Section 5.10 to the value_expression in each select_parameter of the select_list.

   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 Section 5.8, 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 by

   a. Replacing the select_list with the *SQL*_select_list described in Section 5.7.

   b. Inserting an *SQL*_into_clause with a target list as specified in Section 5.7, replacing the into_clause in the statement, if any.

   c. Applying the transformation SQL_{SC} described in Section 5.11 to the search conditions, if any, in the where and having clauses.

   The remainder of the statement is unchanged.

5.4. Cursor Declarations

```
cursor_declaration ::=
    [extended] cursor Ada_identifier_1
        [input_parameter_list]
        for
            query
            [SQL_order_by_clause]
        ;
        [ is cursor_procedures
    end [Ada_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]
```

*Ada*_identifier_1 is the *name* of the cursor. If present, *Ada*_identifier_2 must equal *Ada*_identifier_1.

No two procedures within a cursor may have the same name.

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** must appear in the declaration of the module in which the cursor is declared (see Section 3.2).

Ada Semantics

If a cursor named *C* is declared within an abstract module named *M*, then a subpackage named *C* exists within the Ada package *M* (see Section 5.1). That subpackage contains the declarations of the procedures declared in the sequence cursor_procedures. (*Note*: Some of those procedures may appear by assumption. See Section 5.5.) The text of the procedure declarations is described in Section 5.5.

If the query in a cursor declaration is an extended_query_expression, see Section 3.7.

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 Sections 5.2 and 5.7. 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 $\{SP^j_i\} = \{VE^j_i$ [**as** 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 Section 5.7). 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 Section 5.10) 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 Section 5.10). Then $AdaNAME_i = AdaNAME(VE^j_i)$ for any $j \le m$.

The type of the row record parameter is equivalent in the sense of [1], 3.2.10 and 3.7.2, to a record type whose i^{th} component of the row record parameter, denoted $COMP_{Ada}(SP_i)$, is given by

```
AdaNAME_i : AdaTYPE_i
```

SQL Semantics

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

1. The input_parameter_list and cursor_procedures are discarded, as is the **is** . . . **end** bracket. The cursor name *Ada*_identifier_1 is transformed into SQL_{NAME}(*Ada*_identifier_1).

2. The select_list is transformed into an *SQL*_select_list as described in Section 5.7.

3. The search conditions are transformed using the transform SQL_{SC} of Section 5.11.

The remainder of the declaration is unchanged.

Examples

The following section 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 sections of the language definition.

The example below is a simple cursor declaration.

```
cursor Select_Suppliers
```

```
for
  select SNO, SNAME, SSTATUS, CITY
    from S
;
```

This declaration produces the following Ada code.

```
package Select_Suppliers is          -- 5.4: Ada Semantics

  type Row_Type is                   -- 5.5
    Sno     : Sno_Type;              -- 5.7: Ada Semantics
    Sname   : Sname_Type;
    Sstatus : Sstatus_Type;
    City    : City_Type;
  end record;

  procedure Open;                    -- 5.5: #3

  procedure Fetch (                  -- 5.5: #5
    Row      : in out Row_Type;      -- 5.5: #8 and Ada Semantics #3
    Is_Found : out boolean);         -- 5.5: #5 and Ada Semantics #6

  procedure Close;                   -- 5.5: #4

end Select_Suppliers;
```

The following is an example of a more complex cursor declaration.

```
cursor Supplier_Operations (
  Input_City named Supplier_City     : City not null;
  Adjustment named Status_Adjustment : Sstatus not null)

  for
    select SNO named Supplier_Number,
           SNAME named Supplier_Name,
           SSTATUS + Adjustment named Adjusted_Status,
           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 : Sstatus not null;
      Input_Adjustment named Adjustment : Sstatus) is

      update S
        set SSTATUS = Input_Status + Input_Adjustment
        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       -- 5.4: Ada Semantics

    type Supplier_Row_Record_Type is -- 5.9: Ada Semantics
      Supplier_Number : Sno_Type;    -- 5.7: Ada Semantics
      Supplier_Name   : Sname_Type;
      Adjusted_Status : Sstatus_Type;
      Supplier_City   : City_Type;
    end record;

    procedure Open_Supplier_Operations (
      Supplier_City     : in City_Not_Null;     -- 5.6: Ada Semantics
      Status_Adjustment : in Sstatus_Not_Null); --      #1, #3, Modes

    procedure Fetch_Supplier_Tuple (
      Supplier_Row_Record : in out Supplier_Row_Record_Type;  -- 5.9
      Fetch_Status        : out Operation_Status);            -- 5.13

    procedure Close_Supplier_Operations;

    procedure Update_Supplier_Status (
      Updated_Status : in Sstatus_Not_Null; -- 5.6: Ada Semantics #1
      Adjustment     : in Sstatus_Type);    -- 5.5: Ada Semantics

    procedure Delete_Supplier;

end Supplier_Operations;
```

5.5. Cursor Procedures

```
cursor_procedures ::=
    cursor_procedure {cursor_procedure}

cursor_procedure ::=
    [extended] procedure Ada_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 [Ada_identifier]

fetch_statement ::=
    fetch [Ada_identifier] [into_clause]

close_statement ::=
```

```
    close [Ada_identifier]

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

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

1. *Ada*_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 may 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 Subsection 4.1.9.

6. If *Ada*_identifier is present in an open, fetch, close, cursor_update or cursor_delete statement, then it must 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 Section 5.3) 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
   ```

   See Section 5.9.

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** must 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.

1. 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*.

2. The simple name of P_{Ada} is the name of P.

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 Section 5.6).

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 Section 5.6).

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 Sections 5.2 and 5.4. Let *IC* be the into_clause of 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 Section 5.9.

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 Section 3.7. For extended parameter lists see Section 5.6.

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 Subsections 4.1.9 and 5.13.

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 compilation unit. P_{SQL} has three parts:

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

2. A list of *SQL*_parameter_declarations. An SQLCODE parameter is declared for every SQL procedure. 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 Section 5.6 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 Section 5.6 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 Section 5.7) appear in the parameter declarations of P_{SQL}.

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

3. An *SQL*_SQL_Statement. 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 Section 5.10 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 SQL_{NAME}(*C)* **into** *target list*

where *C* is the cursor name and *target list* as described in Section 5.7.

f. If the statement is an extended_statement, see Section 3.7.

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 Section 5.6) are set from the value of $PARM_{Ada}$(INP) (see Ada semantics above) according to the rule of Section 4.3 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 Section 3.6, is performed.

3. If the value of the SQLCODE parameter is zero, and the statement within the procedure is a fetch_statement, then the value of $COMP_{Ada}(SP_i)$, the i^{th} component of the row record parameter of the procedure, is set from the values of the actual parameters associated with the SQL formal parameters $SQL_{SPNAME}(SP_i)$ and $INDIC_{NAME}(SP_i)$ (see Section 5.7) according to the rule of Section 4.3.

5.6. Input Parameter Lists

Input parameter lists declare the parameters of the procedure or cursor 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 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 ([2], Subsection 4.10.2).

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 ::=
```

```
Ada_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. No two parameters in an input parameter list shall have the same name. 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*) = *Id_2* otherwise, AdaNAME(*INP*) = *Id_1*.

2. For no two parameters, INP_1 and INP_2, in an input parameter list shall AdaNAME(INP_1) = AdaNAME(INP_2).

3. 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 does not support null values, 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 Subsection 4.1.4).

The optional **out** may occur only in an input parameter list that is associated with a procedure or cursor that is extended. The optional **in**, however, may be included (or not) 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, $PARM_{Ada}$(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_declaration $PARM_{SQL}$(INP) is given by

SQL_{NAME}(*Id_1*) DBMS_TYPE(*D*)

where DBMS_TYPE(*D*) is given in Subsection 4.1.4. If **not null** does *not* appear within the textual representation of INP, *and* [*Id_3*.]*Id_4* is the name of a domain that supports null values, then the parameter $INDIC_{SQL}$(INP) is defined and has a textual representation given by the *SQL*_parameter_declaration

$INDIC_{NAME}$(INP) *indicator_type*

where *indicator_type* is the implementation-defined type of indicator parameters ([2], Subsection 5.6.2). The name $INDIC_{NAME}$(INP) does not appear as the name of any other parameter of the enclosing procedure.

5.7. 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 (Subsection 4.2.2).

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

select_parameter ::=
    value_expression [named_phrase] [not_null]
```

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 [2], Section 5.25). Let U_i, for $1 \leq i \leq k$ 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 **.** replaced by a _. 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 Section 4.2) within the order in which the tables were named in the from_clause.

Note: This definition differs from that given in [2], Section 5.25 (4) in specifying that the column references are qualified by table name or correlation name. The table that is described in [2], Section 5.25, paragraph 4 has anonymous columns. The record type being described must have well-defined component names.

Note: 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 a ∗ may cease functioning where a program using a named select list would continue to operate correctly.

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

Ada Semantics

Let SP be a select parameter written as

```
VE [named ID_1] [not null]
```

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

- Let DOMAIN(VE) = *D* (see Section 5.10) where *D* ≠ NO_DOMAIN. If **not null** appears in SP, *or D* is a domain that does not support null values, then AdaTYPE(SP) is the name of the not null-bearing type within the domain *D*; else AdaTYPE(SP) is the null-bearing type within the domain *D*.

- If *Id_1* appears in SP, then AdaNAME(SP) = *Id_1*; else AdaNAME(SP) = AdaNAME(VE) (see Section 5.10).

- No other select parameter SP_i within the select list that contains SP shall be such that AdaNAME(SP_i) = AdaNAME(SP).

SQL Semantics

From a select_list, three SQL fragments must be derived:

1. An *SQL*_select_list within a select statement or cursor declaration.

2. An *SQL*_target_list within a select statement or fetch statement.

3. A list of *SQL*_parameter_declarations.

An *SQL*_select_list is derived from a select_list as follows: any **named** or **not null** phrase within a select parameter is discarded; each value_expression *VE* is replaced by the *SQL*_value_expression SQL_{VE}(*VE*) (see Section 5.10).

The target list generated from a select list is a comma-separated list of *SQL*_target_specifications ([2], Section 5.6). Let SP_i be the select parameter

```
VE_i [named Id_1] [not null]
```

Then the i^{th} *SQL*_target_specification in the *SQL*_target_list is

```
SQL_SPNAME(SP_i) [INDICATOR INDIC_NAME(SP_i)]
```

where $SQL_{SPNAME}(SP_i)$ is an implementation-defined *SQL*_identifier. The INDICATOR phrase appears if the **not null** phrase does *not* appear in the textual representation of SP_i *and* the domain DOMAIN(VE_i) is a domain that supports null values.

The list of *SQL*_parameter_declarations generated by the select_list includes the declarations

```
SQL_SPNAME(SP_i) DBMS_TYPE(DOMAIN(VE))
INDIC_NAME(SP_i) indicator_type
```

where *indicator_type* is the implementation-defined type of indicator parameters ([2], Subsection 5.6.2).

The names $SQL_{SPNAME}(SP_i)$ and $INDIC_{NAME}(SP_i)$ have no appearances within the procedure other than those mentioned.

5.8. Value Lists and Column Lists

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

insert_column_specification ::=
    column_specification [named_phrase] [not_null]

insert_value_list ::=
    insert_value {, insert_value}

insert_value ::=
    null | constant_reference |
    literal | column_specification
```

Each column_specification 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 Section 5.3. See also [2], Section 8.7(3).)

Let *C* be the insert_column_specification

```
Col [named Id] [not null]
```

Then AdaNAME(*C*) is defined to be *Id*, if *Id* is present; otherwise it is *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 domain that does not support null values, 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. The *target domain* of V_i is DOMAIN(C_i), and *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 $1 \le i \le m$, if V_i is

   a. **null**, then DOMAIN(C_i) shall be a domain that supports null values.

   b. A literal or reference to the constant *k*, and each V_i conforms to DOMAIN(C_i) (see Section 3.5).

   c. An *SQL*_column_name, then it shall be identical to the *SQL*_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 Section 5.2 on the basis of the functions AdaNAME and AdaTYPE described above. For the name of the record type and its place of declaration, see Section 5.9.

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 Section 5.2.

SQL Semantics

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

`Col`$_i$ [**named** *Id*$_i$] `[not null`$_i$`]`

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

`SQL`$_{\text{Col}}$`(`C_i`)` *data_type*$_i$

appear in the list of SQL parameter declarations, where

1. $\text{SQL}_{\text{Col}}(C_i)$ is an implementation-defined *SQL*_identifier that appears nowhere else.

2. *data_type*$_i$ is the SQL data type known to the environment for the domain DOMAIN(*Col*$_i$). See Subsection 4.1.4.

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

`INDIC`$_{\text{NAME}}$`(`C_i`)` *indicator_type*

also appears in the list of SQL parameter declarations, where

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

2. *indicator_type* is the implementation-defined type of indicator parameters ([2], Subsection 5.6.2).

An insert_column_list and insert_value_list pair are transformed into an *SQL*_insert_column_list and *SQL*_insert_value_list pair 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.

2. An insert_value_list is transformed into an *SQL*_insert_value_list 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 $SQL_{VE}(k)$ (see Sections 4.3 and 5.10).

c. An *SQL*_column_name C_i is replaced by

$SQL_{Col}(C_i)$ [INDICATOR $INDIC_{NAME}(C_i)$]

where the INDICATOR phrase appears whenever the indicator parameter, $INDIC_{NAME}(C_i)$, is defined (see above).

5.9. 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 [Ada_identifier_1] [: record_id]

insert_from_clause ::=
    from [Ada_identifier_1] [: record_id]

record_id ::=
    new Ada_identifier_2 | record_reference
```

At least one Ada_identifier or record_id must appear in an into_clause or insert_from_clause.

Ada Semantics

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

1. If *Ada*_identifier_1 appears in *IC*, then $PARM_{Row}(IC)$ = *Ada*_identifier_1.

2. Otherwise, if record_id is a record_reference referencing the record declaration *R*, then $PARM_{Row}(IC)$ = AdaNAME(*R*). See Subsection 4.1.6.

3. Otherwise, record_id takes the form

   new *Ada*_identifier_2

   and $PARM_{Row}(IC)$ = *Row*.

Define $TYPE_{Row}(IC)$ as follows:

1. If record_id has the form

   new *Ada*_identifier_2

   then $TYPE_{Row}(IC)$ = *Ada*_identifier_2. In this case, a record type declaration with name *Ada*_identifier_2 will appear in the declarative region immediately preceding the Ada declaration of the procedure within which the into or insert_from clause appears.
   Note: Thus, for fetch statements, the type declaration appears within the cursor sub-package. (See Section 5.4.)

2. Otherwise, $TYPE_{Row}(IC)$ is the record type referenced by the record_reference.

Note: 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** *Ada*_identifier" (see Sections 5.4, 5.2, and 5.7).

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 Subsections 4.1.7, 4.1.6, and 4.1.9 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 Subsection 4.2.3.

The two examples below illustrate the use of a previously declared record object in the **into** clauses of select statements. The two examples illustrate a possible scenario where an SQL module might contain two select statements for the same object, namely a part. The first select statement below exists in a cursor declaration because it has the potential to return more than one record. The second select statement exists in a procedure because it can return at most one record from the table. Since both select statements retrieve the same type of object from the database, they may share a row record. The row record contains the definition of the part abstraction. To share a record object, declare the record first, and then reference it in the **into** clauses of both select statements.

```
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 * 16 named Weight_In_Ounces,
           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 * 16 named Weight_In_Ounces,
         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_Not_Null); -- 5.2: Ada Semantics

  procedure Fetch (
    Parts_By_City_Row : in out Parts_Row_Record_Type;
                                    -- 5.5: Ada Semantics #3
    Is_Found          : out boolean);   -- 5.5: Ada Semantics #6

  procedure Close;

end Parts_By_City;

procedure Parts_By_Number (
  Part_Number            : in Pno_Not_Null; -- 5.2: 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 parameters, types, and names of the generated row record parameter.

```
procedure Part_Name_By_Number (
  Input_Pno named Part_Number : Pno 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 -- 5.9: Ada Semantics
  Part_Name : Pname_Type;               -- 4.1.6
end record;

procedure Part_Name_By_Number(
  Part_Number              : in Pno_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, SSTATUS, CITY)
  values
  status Operation_Map named Insert_Status
;
```

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

```
type Row_Type is record                -- 5.3: #1
  Sno     : Sno_Type;                  -- 5.2: Ada Semantics,
  Sname   : Sname_Type;                --    Parameter Profiles, #3
  Sstatus : Sstatus_Type;              -- 5.8: Ada Semantics
  City    : City_Type;
end record;

procedure Add_To_Suppliers (
  Row           : in 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, meaning that 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', 'LT 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);
```

5.10. 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 defined either in a definitional module or in the current abstract module.

2. Value expressions are strongly typed; therefore, a domain conversion operation must be 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_specification |
    input_reference|
    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_specification)

all_set_function ::=
    {AVG | MAX | MIN | SUM} ([ALL] value_expression)
    see [2] 5.8

domain_conversion ::=
    domain_reference ( value_expression )
```

Four mappings are defined on value_expressions: AdaNAME, DOMAIN, DATACLASS, 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. (*Note*: If DOMAIN is not defined on a value expression, then the value expression is not legal.) The class of domains is augmented by the special value NO_DOMAIN, the domain of literals and universal constants.

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 (Section 2.4).

The mapping SCALE returns the scale of the result 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, 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, and DATACLASS(*L*) is defined in Section 2.4. If *L* is a numeric literal, then SCALE(*L*) is defined in Section 2.4.

2. **References**. Let F be an input_reference, constant_reference, domain_parameter_reference, or column_specification; let G be the object to which F makes reference. Then DOMAIN(F)=DOMAIN(G), SCALE(F)=SCALE(G), and DATACLASS(F)=DATACLASS(G). AdaNAME(F) = F if F is the simple name of G; otherwise, AdaNAME(F) = NO_NAME.

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*)), SCALE(*SF(VE)*) = SCALE(*VE*) and DATACLASS(*SF(VE)*) = DATACLASS(*VE*).

   - If *SF* is COUNT, then DOMAIN(COUNT(*VE*)) = DOMAIN(COUNT(∗)) = *NO_DOMAIN* and DATACLASS(COUNT(*VE*)) = DATACLASS(COUNT(∗)) = **integer** and SCALE(COUNT(*VE*)) = SCALE(COUNT(∗)) = 0 (see Section 2.4).

   - If *SF* is SUM, then DOMAIN(SF(*VE*)) = *NO_DOMAIN*, DATACLASS(SF(*VE*)) = DATACLASS(*VE*), and SCALE(SF(*VE*)) = SCALE(*VE*) ([2], Section 5.8 (9)).

   - If *SF* is AVG, then DOMAIN(SF(*VE*)) = *NO_DOMAIN* and DATACLASS(SF(*VE*)) and SCALE(SF(*VE*)) are implementation defined ([2] 5.8, Section (9)).

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

        i. If SCALE(D) < SCALE(VE) then SCALE(*D(VE)*) = SCALE(*VE*) *and* a warning message must 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 Section 3.5). A conversion is allowable if DATACLASS(VE) is float and DATACLASS(D) is either fixed or integer.

        ii. Otherwise SCALE(*D(VE)*) = SCALE(*D*).

     b. DATACLASS(*D*) and DATACLASS(*VE*) are both character.

     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 [1], Section 4.7.)

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

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

DOMAIN(VE_1) = D_1;
DOMAIN(VE_2) = D_2;
DATACLASS(VE_1) = T_1;
DATACLASS(VE_2) = T_2;
SCALE(VE_1) = S_1;
SCALE(VE_2) = S_2;

Then

a. For unary operators (+, -)

- AdaNAME(*[+|-]*VE_1) = NO_NAME.
- DOMAIN(*[+|-]*VE_1) = D_1 *provided* that T_1 is a numeric data class;
- DATACLASS(*[+|-]*VE_1) = T_1;
- SCALE(*[+|-]*VE_1) = S_1;

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

c. If T_1 and T_2 are numeric, then DATACLASS(VE_1 *op* VE_2) = max(T_1, 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

- Both T_1 and T_2 are numeric data classes, and either
  - $D_1 \neq$ 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 [2], Section 5.9(4,5).)

Examples

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.
- AdaNAME(*1+1.0*) = NO_NAME, DOMAIN(*1+1.0*) = NO_DOMAIN, and DATACLASS(*1+1.0*) = **fixed**, as DATACLASS(1) = **integer** and DATACLASS(1.0) = **fixed**.
- 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 [2], Section 5.8).
- 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 [2], Section 5.8).
- The result of an AVG set function is treated as though it were a literal of an implementation-defined data class and scale (see [2], Section 5.8).

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 Section 4.3). 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 specification 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, then $SQL_{VE}(VE) = SQL_{VE}(E)$.

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

      constant `C [: D [`**not null**`]]` **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, then $SQL_{VE}(VE) = SQL_{VE}(E)$.

   e. If *VE* is an input parameter, then $SQL_{VE}(VE)$ is $SQL_{NAME}(VE)$ [INDICATOR $INDIC_{NAME}$(VE)], where INDICATOR $INDIC_{NAME}$(VE) appears precisely when $INDIC_{NAME}$(VE) is defined. See Section 5.6.

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.

5.11. Search Conditions

The concrete syntax of search conditions differs from that of SQL only in that value_expression (Section 5.10) replaces *SQL*_value_expression in the definition of the atomic predicates ([2], Sections 5.11 through 5.17). 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 (Section 5.10). For an atomic predicate to be well formed, then for any pair of distinct i and j,

1. If DOMAIN(OP_i) ≠ NO_DOMAIN and DOMAIN(OP_j) ≠ 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 such that

      i. DOMAIN(OP_k) ≠ NO_DOMAIN, and

      ii. For all l, 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 Section 5.10, to the value expressions appearing within the search condition and the transformation SQL_{SQ}, defined in Section 5.12, 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 [2], Section 5.18).

2. If P is of the form: **not** P_1, then $SQL_{SC}(P)$ is **not** $SQL_{SC}(P_1)$. (See [2], Section 5.18).

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 [2], Section 5.11).

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 [2], Section 5.12).

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 [2], Section 5.13).

6. If P is of the form: VE_1 **[not] like** VE_2 **escape** c where *c* is a character, then
   $SQL_{SC}(P)=SQL_{VE}(VE_1)$ **[not] like** $SQL_{VE}(VE_2)$ **escape** c
   (See [2], Section 5.14).

7. If P is of the form: C **is [not] null** where C is an *SQL*_column_specification, then $SQL_{SC}(P)=C$ **is [not] null**. (See [2], Section 5.15). *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 [2], Section 5.16).

9. If P is of the form: **exists** SQ, then $SQL_{SC}(P)=$**exists** $SQL_{SQ}(SQ)$. (See [2], Section 5.17).

5.12. Subqueries

The concrete syntax of a subquery ([2], Section 5.24) differs from that of query specification ([2], Section 5.25) in that the select list is limited to at most one parameter. Further, that parameter, when present, takes the form of a value_expression (Section 5.10), not a select_parameter (Section 5.7), 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*). *DOMAIN*(*SQ*) is 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 applying the transformation SQL_{SC} to the search_conditions in the **where** and **having** clauses, if present.

5.13. Status Clauses

A status clause serves to attach a status mapping 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** *(*L_1*, . . . ,* L_n*);*

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

(see Subsection 4.1.9),

then:

1. The procedure P_{Ada} (Sections 5.3 and 5.5) 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 Subsection 4.1.9).

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

References

[1] *Reference Manual for the Ada Programming Language*
Ada Joint Program Office, 1983.

[2] *Database Language - SQL*
American National Standards Institute, 1989.
X3.135-1989.

[3] *Database Language - Embedded SQL X3,168-1989*
American National Standards Institute, 1989.

[4] Graham, Marc H.
Guidelines for the Use of the SAME.
Technical Report CMU/SEI-89-TR-16, ADA215846, Software Engineering Institute, May 1989.

Appendix A: Transform Chart

.

.

.

Appendix B: 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 SQL manual (Section 5.7).

Cursor. See SQL manual (Section 8.3).

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) 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) 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. The SAMeDL notion that corresponds to the SQL SCHEMA.

SQL. Structured Query Language.

SQLCODE. See SQL manual (Section 7.3).

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.

Appendix C: SAMeDL Standard Modules and Support Packages

This appendix contains the predefined SAMeDL definitional modules SAMeDL_Standard and SAMeDL_System. It also contains a brief description of the SAME standard support packages along with their Ada package specifications.

C.1. 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 status map Standard_Map and for the standard base domains.

```
definition module SAMeDL_Standard is

    exception SQL_Database_Error;

    -- standard status map
    status Standard_Map named Is_Found uses boolean is
        (0 => True, 100 => False);

    -- SQL_Int is based on the Ada type SQL_Standard.Int
    base domain SQL_Int is
        first : integer;
        last  : integer;
    end SQL_Int;

    base domain body SQL_Int 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);'

        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);'

        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);'

        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 is
    first : integer;
    last  : integer;
end SQL_Smallint;

base domain body SQL_Smallint 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);'

    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);'

    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);'

    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 is
    first : float;
    last  : float;
```

```
end SQL_Real;

base domain body SQL_Real 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);'

    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);'

    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);'

    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 is
    first : float;
    last  : float;
end SQL_Double_Precision;

base domain body SQL_Double_Precision 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);'

    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);'
```

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);'
```

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**
```
    length : integer;
```
end `SQL_Char;`

base domain body `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]_Type, [self]_Not_Null);'
```

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]_Type, [self]_Not_Null);'
```

subdomain pattern is

```
   'subtype [self]_Not_Null is [parent]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]_Type, [self]_Not_Null);'
```

```
    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
enumeration base domain SQL_Enumeration_As_Int is
    map := pos;
end SQL_Enumeration_As_Int;

base domain body SQL_Enumeration_As_Int is

    domain pattern is

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

    derived domain pattern is

    'type [self]_Type is new [parent]_Type;'

    subdomain pattern is

    'subtype [self]_Type is [parent]_Type;'

    for not null type name use '[self]';
    for null type name use '[self]_Type';
    for data class use enumeration;
    for dbms type use integer;
    for conversion from dbms to not null use function
        '''val';
    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';
    for conversion from not null to dbms use function
        '''pos';

end SQL_Enumeration_As_Int;

-- SQL_Enumeration_As_Char is based on the Ada type
-- SQL_Standard.Char
enumeration base domain SQL_Enumeration_As_Char is
    map := image;
end SQL_Enumeration_As_Char;

base domain body SQL_Enumeration_As_Char is

    domain pattern is

    'package [self]_Pkg is new SQL_Enumeration_Pkg('
```

```
            '[enumeration]);'
        'type [self]_Type is new [self]_Pkg.SQL_Enumeration;'

        derived domain pattern is

        'type [self]_Type is new [parent]_Type;'

        subdomain pattern is

        'subtype [self]_Type is [parent]_Type;'

        for not null type name use '[self]';
        for null type name use '[self]_Type';
        for data class use enumeration;
        for dbms type use character '([length])';
        for conversion from dbms to not null use function
            '''value';
        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';
        for conversion from not null to dbms use function
            '''image';

    end SQL_Enumeration_As_Char;

end SAMeDL_Standard;
```

C.2. 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;

        -- SQL Standard value for successful execution of an SQL DML
        -- statement
        constant Success is 0;
        -- SQL Standard value for data not found
        constant Not_Found is 100;

    end SAMeDL_System;
```

C.3. Standard Support Packages

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.3.1. 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 Section C.3.2.1; [3]) 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.3.1.1. 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.3.2.3), which implements three-valued logic (see [4]). 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.3.1.2. 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

[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.

operation. The unary operations that are provided are +, -, and *Abs*. The binary operations include +, -, ∗, and /. Finally, all numeric domains provide the exponentiation operation (∗∗).

C.3.1.3. 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.

C.3.1.4. 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 of SQL2. 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.3.1.1 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.3.1.5. 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.3.1.3 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.3.1.6. 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 [2]. 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.3.1.7. 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.3.2. Standard Support Package Specifications

C.3.2.1. SQL_Standard

The package SQL_Standard is part of the ANSI standard for embedded SQL [3].

```
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 Smallint is range bs..ts;
    type Int is range bi..ti;
    type Real is digits dr;
    type Double_Precision is digits dd;
    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;
    subtype Indicator_Type is t;

-- 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.

end SQL_Standard;
```

C.3.2.2. SQL_Exceptions

```
package SQL_Exceptions is

    Null_Value_Error : exception;

end SQL_Exceptions;
```

C.3.2.3. 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.3.2.4. 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.3.2.5. 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.3.2.6. 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.3.2.7. 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.3.2.8. 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.3.2.9. 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;
```

Index

Table of Contents