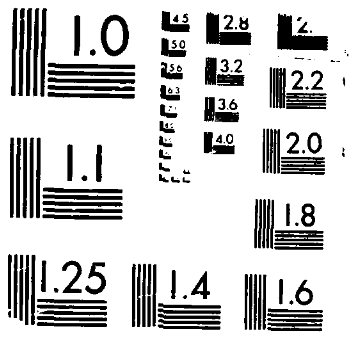
AD-A194 480 LEVEL 1 ADA/SQL (STRUCTURED QUERY LANGUAGE) DATABASE 1/2
LANGUAGE INTERFACE U..(U) INSTITUTE FOR DEFENSE
ANALYSES ALEXANDRIA VA B BRYKCZYNSKI ET AL. SEP 87
UNCLASSIFIED IDA-M-360 IDA/HQ-87-32693 MDA903-84-C-0031 F/G 12/5 NL



MICROCOPY RESOLUTION TEST CHART

·UREAU ·· STANDARDS 1963 A

UNCLASSIFIED

Copy 12 of 55 copies

(2)

AD-A194 480

DTIC FILE COPY

IDA MEMORANDUM REPORT M-360

LEVEL 1 Ada/SQL DATABASE LANGUAGE INTERFACE USER'S GUIDE

DTIC
S ELECTE D
JUN 2 8 1988
D

Bill Brykczynski
Fred Friedman
Kerry Hilliard
Audrey Hook

September 1987

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

Prepared for
Office of the Under Secretary of Defense for Research and Engineering

IDA

INSTITUTE FOR DEFENSE ANALYSES
1801 N. Beauregard Street, Alexandria, Virginia 22311

UNCLASSIFIED

IDA Log No. HQ 87-32693

88 [illegible]

DEFINITIONS

IDA publishes the following documents to report the results of its work.

Reports

Reports are the most authoritative and most carefully considered products IDA publishes. They normally embody results of major projects which (a) have a direct bearing on decisions affecting major programs, or (b) address issues of significant concern to the Executive Branch, the Congress and/or the public, or (c) address issues that have significant economic implications. *IDA Reports are reviewed by outside panels of experts to ensure their high quality and relevance to the problems studied, and they are released by the President of IDA.*

Papers

Papers normally address relatively restricted technical or policy issues. They communicate the results of special analyses, interim reports or phases of a task, ad hoc or quick reaction work. Papers are reviewed to ensure that they meet standards similar to those expected of refereed papers in professional journals.

Memorandum Reports

IDA Memorandum Reports are used for the convenience of the sponsors or the analysts to *record substantive work done in quick reaction studies and major interactive technical support* activities; to make available preliminary and tentative results of analyses or of working group and panel activities; to forward information that is essentially unanalyzed and unevaluated; or to make a record of conferences, meetings, or briefings, or of data developed in the course of an investigation. Review of Memorandum Reports is suited to their content and intended use.

The results of IDA work are also conveyed by briefings and informal memoranda to sponsors and others designated by the sponsors, when appropriate.

The work reported in this document was conducted under contract MDA 903 84 C 0031 for the Department of Defense. The publication of this IDA document does not indicate endorsement by the Department of Defense, nor should the contents be construed as reflecting the official position of that agency.

This Memorandum Report is published in order to make available the material it contains for the use and convenience of interested parties. The material has not necessarily been completely evaluated and analyzed, nor subjected to IDA review.

Approved for public release: distribution unlimited.

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE AD-A194480

1a REPORT SECURITY CLASSIFICATION Unclassified	1b RESTRICTIVE MARKINGS
2a SECURITY CLASSIFICATION AUTHORITY 2b DECLASSIFICATION/DOWNGRADING SCHEDULE	3 DISTRIBUTION/AVAILABILITY OF REPORT Public release/unlimited distribution.
4 PERFORMING ORGANIZATION REPORT NUMBER(S) IDA Memorandum Report M-360	5 MONITORING ORGANIZATION REPORT NUMBER(S)
6a NAME OF PERFORMING ORGANIZATION Institute for Defense Analyses · 6b OFFICE SYMBOL IDA	7a NAME OF MONITORING ORGANIZATION OUSDA, DIMO
6c ADDRESS (City, State, and Zip Code) 1801 N. Beauregard St. Alexandria, VA 22311	7b ADDRESS (City, State, and Zip Code) 1801 N. Beauregard St. Alexandria, VA 22311
8a NAME OF FUNDING/SPONSORING ORGANIZATION Defense Logistics Agency · 8b OFFICE SYMBOL (if applicable) DLA-ZWS	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER MDA 903 84 C 0031
8c ADDRESS (City, State, and Zip Code) Cameron Station Alexandria, VA 22304-6100	10 SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. · PROJECT NO. · TASK NO. T-T5-423 · WORK UNIT ACCESSION NO.

11 TITLE (Include Security Classification)
Level 1 Ada/SQL Database Language Interface User's Guide (U)

12 PERSONAL AUTHOR(S)
Bill Brykczynski, Fred Friedman, Kerry Hilliard, Audrey A. Hook

13a TYPE OF REPORT	13b TIME COVERED	14 DATE OF REPORT (Year, Month, Day)	15 PAGE COUNT
Final	FROM ______ TO ______	1987 September	108

16 SUPPLEMENTARY NOTATION
((Structured query language).

17 COSATI CODES			18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Ada programming language; structured query language (SQL); binding specifications; relational database systems; software engineering; database tables; interface specifications; administrative processing systems.

19 ABSTRACT (Continue on reverse if necessary and identify by block number)

The purpose of this IDA Memorandum Report is to present a binding specification between the programming language Ada and the database language SQL. This document will be used to provide a capability for accessing a relational database from the Ada language and will provide a cross reference between the Ada/SQL language and the SQL language. As a user's guide, M-360 addresses the functionality of the ANSI SQL standard and identifies a subset implementation of it as Level 1. A Level 1 implementation was delivered to the DLA for use with a database package that is intended for regional administrative systems. An Ada application prototype will be implemented to demonstrate software engineering using Ada in the environment provided for these regional adminstrative systems. In addition, an IDA Memorandum Report M-361 has been published as a companion document which provides the source listings for the demonstration software that implements this Level 1 version of Ada/SQL.

20 DISTRIBUTION/AVAILABILITY OF ABSTRACT [✓ UNCLASSIFIED/UNLIMITED] ☐ SAME AS RPT. ☐ DTIC USERS	21 ABSTRACT SECURITY CLASSIFICATION Unclassified	
22a NAME OF RESPONSIBLE INDIVIDUAL Audrey A. Hook	22b TELEPHONE (Include area code) (703) 824-5501	22c OFFICE SYMBOL IDA/CSED

DD FORM 1473, 84 MAR
83 APR edition may be used until exhausted
All other editions are obsolete
SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

UNCLASSIFIED

IDA MEMORANDUM REPORT M-360

LEVEL 1 Ada/SQL DATABASE LANGUAGE INTERFACE USER'S GUIDE

Bill Brykczynski
Fred Friedman
Kerry Hilliard
Audrey Hook

September 1987

Accesion For		
NTIS CRA&I		✓
DTIC TAB		☐
Unannounced		☐
Justification		
By		
Distribution /		
Availability Codes		
Dist	Avail and / or Special	
A-1		

INSTITUTE FOR DEFENSE ANALYSES

Contract MDA 903 84 C 0031
Task T-T5-423

UNCLASSIFIED

Table of Contents

Preface

The purpose of IDA Memorandum Report M-360, Level 1 Ada/SQL Database Language Interface, is to forward data developed in the course of an investigation. This Memorandum presents a binding specification between the programming language Ada and the database language SQL. In particular, this specification is in the exact format which is found in the SQL specification (i.e. chapter names are identical)

The importance of this document is based on fulfilling the objective of Task Order T-T5-423, Defense Logistics Agency Information Systems, which is to provide a capability for accessing a relational database from the Ada language. M-360 will be used to provide a cross reference between the Ada/SQL language and the SQL language. As a Memorandum Report, M-360 is directed to those users of Ada/SQL within the Defense Logistics Agency.

1. Scope

This report has been prepared in partial fulfillment of IDA task T-T5-423, "Defense Logistics Agency Information Systems." This User's Guide for the Ada/SQL Database Language Interface addresses the functionality of the ANSI SQL standard but identifies a sub-set implementation of it as Level 1. A Level 1 implementation was delivered to DLA for use with a database package that is intended for regional administrative systems. An Ada application prototype will be implemented to demonstrate software engineering using Ada in the environment provided for these regional administrative systems.

IDA Memorandum 361 is a companion document that provides the source listings for the demonstration software that implements this Level 1 version of Ada/SQL.

This User's Guide specifies the syntax and semantics of Level 1 Ada/SQL. The Ada/SQL DDL is used for declaring the structures and integrity constraints of database tables that will be used with Ada/SQL DML to manipulate the database tables from within an Ada program.

DRAFT

2. References

1) American National Standard "Database Language SQL", ISO/DIS 9075 ANSI X3.135-1986, dated May 1986

2) Military Standard Ada Programming Language, ANSI/MIL-STD-1815A, dated January 1983

3. Overview

3.1. Organization

1) Clause 3.2 "Notation" defines the syntactic notation used in the manual.

2) Clause 3.3 "Conventions" define the terms used to specify the syntactic elements in this manual.

3) Clause 4 "Concepts" defines terms and presents concepts used in the definition of Ada/SQL, explains why limitations were placed on certain Ada concepts and discusses possible future enhancements.

4) Clause 5 "Common Elements" defines language elements that occur in several parts of Ada/SQL.

5) Clause 6 "Schema Definition Language" defines in detail the design of the DDL used for specifying database tables.

6) Clause 7 "Program Environment" defines the program environment required for an Ada/SQL program.

7) Clause 8 "Data Manipulation Language" defines the statements uses in Ada/SQL to manipulate data within the databases.

3.2. Notation

The syntactic notation used in this document is a blend of the simple variants of BNF ("Backus Normal Form" or "Backus-Naur Form") used in the ANSI Standard SQL Document and the Ada Programming Language Military Standard. The extensions noted below augment BNF. The syntax is as follows:

Square brackets ([]) indicate optional elements.

Ellipses (...) indicate elements that may be repeated one or more times.

Braces ({ }) group sequences of elements.

Boldface words are used to denote reserved words.

A vertical bar separates alternative items unless it occurs immediately after an opening brace, in which case it stands for itself.

If the name of any syntactic category starts with an underlined prefix, it is equivalent to the category name without the prefix. The prefix is intended to convey some semantic information.

In the BNF syntax, a production symbol <A> is defined to "contain" a production symbol <B> if <B> occurs someplace in the expansion of <A>. If <A> contains <B>, then <B> is "contained in" <A>. If <A> contains <B>, then <A> is the "containing" <A> production symbol for <B>.

3.3. Conventions

Syntactic elements of this manual are specified in terms of:

1) **Function**: A short statement of the purpose of the element.

2) **Format**: A BNF definition of the syntax of the element.

3) **Syntax Rules**: Additional syntactic constraints not expressed in BNF that the element shall satisfy.

4) ***Level 1 Implementation Rules***: *Syntactic constraints specific to Level 1 Ada/SQL that the element shall*

satisfy.

5) **General Rules**: A sequential specification of the run-time effect of the element.

In the Syntax Rules, the term "shall" defines conditions that are required to be true of syntactically conforming Ada/SQL language.

In the Level 1 Implementation Rules, the term "shall" defines conditions that are required to be true in Ada/SQL Level 1, but are expected to change in future levels.

In the General Rules, the term "shall" defines conditions that are tested at run-time during the execution of Ada/SQL statements. If all such conditions are true, then the statement executes successfully. If any such condition is false, then the statement does not execute successfully and the statement execution has no effect on the database, the SQLCODE parameter is set and an exception may be raised. *Note: Level 1 Ada/SQL will not set SQLCODE, but will raise exceptions.*

The elements described in this manual for Ada/SQL are standard Ada elements. Not all features of an Ada element may be permitted in conjunction with Ada/SQL statements. Limitation of an element in Ada/SQL only limits its use when used in direct association with the Ada/SQL portions of a compilation unit.

4. Concepts

4.1. Sets

A set is an unordered collection of distinct objects.

A multi-set is an unordered collection of objects that are not necessarily distinct.

A sequence is an ordered collection of objects that are not necessarily distinct.

The cardinality of a collection is the number of objects in that collection. Unless specified otherwise, any collection may be empty.

4.2. Data Types

A data type is a set of representable values. The logical representation of a value is a <literal>. The physical representation of a value is implementor-defined. The implementor-defined representation of values will have no affect on programs and/or the use of the Ada/SQL system, providing programs use only those logical operations on data that are representation-independent. Hence, Ada/SQL programs may be fully transportable.

A value is primitive, in that it has no logical subdivision within this specification. A value is a null value or a nonnull value.

A null value is an implementor-defined type-dependent special value that is distinct from all nonnull values of that type.

A non-null value may be of any data type defined for Ada/SQL: character string (one dimensional array of characters), integer, floating point number, and enumeration, the value of any program object, of a permitted type, may be stored within a database column.

4.2.1. Character Strings

A character string consists of a sequence of characters of the predefined type CHARACTER defined in the Ada STANDARD package. A character string has a length, which is a positive integer that specifies the number of characters in the sequence.

Character strings may be defined as STRINGs or as any single-dimension array of CHARACTER with integer index. Character strings are the only Ada/SQL data types defined as Ada arrays.

Character strings are comparable in <Ada/SQL statement>s only if they are of the same Ada type. A character string is identical to another character string if and only if it is equal to that character string in accordance with the comparison rules specified in 5.11, "<comparison predicate>".

A character string value shall only be assigned to a data item or program variable of the same type.

4.2.2. Numbers

A number is either an exact numeric value (Ada integer types) or an approximate numeric value (Ada floating point types). *Ada fixed point types are not supported in Level 1.*

An exact numeric value is a whole number, either positive, negative or zero. An exact numeric value can be constrained to fall within a range.

An approximate numeric value consists of a mantissa and an exponent. The mantissa is a signed numeric value, and the exponent is a signed integer that specifies the magnitude of the mantissa. An approximate numeric value has a precision. The precision is a positive integer that specifies the number of significant decimal digits in the mantissa. An approximate numeric value can be constrained to fall within a range.

Numbers are comparable in <Ada/SQL statement>s only if they are of the same Ada type. A number is identical to another number if and only if it is equal to that number in accordance with the comparison rules specified in 5.11, "<comparison predicate>".

A number value shall only be assigned to a data item or program variable of the same type.

4.2.3. Enumeration Types

Enumeration types as defined by Ada syntax are permitted. Enumeration types are characterized by a sequence of values which define ordered sets in which possible values are explicitly listed. The predefined enumeration types CHARACTER and BOOLEAN will be recognized. CHARACTER is an enumeration type with values of the 128 characters of the ASCII code. BOOLEAN is an enumeration type with values of TRUE and FALSE. The Ada/SQL operations available on enumeration types are the same as are available on strings, except that LIKE is not available for enumeration types. *The representation clause for enumeration types is not permitted in Level 1.*

Enumeration types will be represented in the database such that their Ada sort order is preserved.

Enumeration values are comparable in <Ada/SQL statement>s only if they are of the same Ada type. An enumeration value is identical to another enumeration value if and only if it is equal to that enumeration value in accordance with the comparison rules specified in 5.11, "<comparison predicate>".

An enumeration value shall only be assigned to a data item or program variable of the same type.

4.3. Columns

A column is a multi-set of values that may vary over time. All values of the same column are of the same data type and are values in the same table. A value of a column is the smallest unit of data that can be selected from a table and the smallest unit of data that can be updated.

A column has a description and an ordinal position within a table. The description of a column includes its data type (which includes range constraints) and indication of whether the column is constrained to contain only nonnull or nonnull/unique values. The description of a character string column includes its length. The description of a floating point numeric column includes the number of digits of precision and upper and lower numeric bounds. The description of an integer numeric column includes upper and lower numeric bounds. The description of an enumeration column includes the enumeration literals and their order.

A named column is a column of a named table or a column that inherits the description of a named column. The description of a named column includes its name.

4.4. Tables

A table is a multi-set of rows. A row is a nonempty sequence of values. Every row of the same table has the same cardinality and contains a value of every column of that table. The n^{th} value in every row of a table is a value of the n^{th} column of that table. The row is the smallest unit of data that can be inserted into a table and deleted from a table.

The degree of a table is the number of columns of that table. At any time, the degree of a table is the same as the cardinality of each of its rows and the cardinality of a table is the same as the cardinality of each of its columns.

A table has a description. The description includes a description of each of its columns.

A base table is a named table defined by a <table definition>. The description of a base table includes its name.

A derived table is a table derived directly or indirectly from one or more other tables by the evaluation of a <query specification>. The values of a derived table are those of the underlying tables when it is derived.

A viewed table is a named derived table. *In Level 1 Ada/SQL, a viewed table is defined with a <table definition>, just as is a base table. The actual view definition must also be defined to the underlying database using the DBMS DDL. The description of a viewed table includes its name. Future levels of Ada/SQL will include the capability to specify <view definition>s.*

A table is either updatable or read-only. The operations of insert, update, and delete are permitted for updatable tables and are not permitted for read only tables.

A grouped table is a set of groups derived during the evaluation of a <group by clause>. A group is a multi-set of rows in which all values of the grouping column(s) are equal. A grouped table may be considered as a collection of tables. Set functions may operate on the individual tables within the grouped table.

A grouped view is a viewed table derived from a grouped table.

4.5. Integrity Constraints

Integrity constraints define the valid states of the database by constraining the values in the base tables. Constraints may be defined to prevent two rows in a table from having the same values in a specified column or columns (_NOT_NULL_UNIQUE) or to prevent a column from containing a null value (_NOT_NULL). *For Level 1 Ada/SQL multiple column unique constraints must be defined in the underlying DBMS DDL only. There is no way to define multiple column uniqueness in Level 1 Ada/SQL.*

The integrity constraints _NOT_NULL and _NOT_NULL_UNIQUE are effectively checked after execution of each <Ada/SQL statement>. If the base table associated with a _NOT_NULL or _NOT_NULL_UNIQUE integrity constraint does not satisfy that integrity constraint, then the <Ada/SQL statement> has no effect, and the appropriate error indication is returned.

The integrity constraints _NOT_NULL and _NOT_NULL_UNIQUE must be placed on columns of a base table (through the DDL of the underlying DBMS) for the constraint to be adhered to. These constraints shall also be placed on the appropriate columns when tables are defined in the Ada/SQL DDL. However only the constraints placed through the underlying DBMS will be checked after the execution of statements.

Numeric and enumeration values which may be contained in a column may be given range constraints, indicating the low value and high value permitted in that column in the database.

Range constraints are placed on columns of database tables through table definitions in the Ada/SQL DDL. The underlying DBMS may or may not allow such constraints to be placed on columns via its DDL.

When using a program value, to directly set the value of a column in a database table, range constraints are effectively checked after execution of each <Ada/SQL statement>. If the program value does not satisfy the range constraint the <Ada/SQL statement> has no effect on the table. *Range constraints will not be checked in Level 1 Ada/SQL.*

Range constraints will not be checked if the value set in the constrained column is not strictly a program value, i.e. if it is generated within the database, unless the underlying DBMS has the ability to restrict columns in such a way.

Range constraints will be checked for all columns when they are retrieved from the database. If the retrieved value does not satisfy the column's range constraint an exception will be noted. *Range constraints will not be checked in Level 1 Ada/SQL.*

4.6. Schemas

A <schema> is a persistent object specified by the Ada/SQL schema definition language (Ada/SQL DDL). It consists of all <table definition>s known to the system for a specified <authorization identifier> in an environment. The "environment" for an Ada/SQL system includes all <schema>s that may be accessed through it.

The tables defined by a <schema> are considered to be "owned by" the <authorization identifier> specified for that <schema>. *In Level 1 Ada/SQL, tables and views must be created, dropped or altered and privileges defined outside of the Ada/SQL environment using the DDL appropriate for the underlying DBMS; several parts of the required DDL are not defined for Level 1 Ada/SQL and no automated tool is supplied to translate Ada/SQL DDL to the DDL of the underlying DBMS.*

4.7. The Database

The database is the collection of all data defined by the <schema>s in an environment.

4.8. Program Environment

An Ada/SQL program environment is an Ada application program, possibly consisting of multiple subprograms, using Ada/SQL DDL to define one or more databases and Ada/SQL DML to manipulate data in the defined databases. An Ada/SQL program environment consists of zero or more cursors specified by <declare cursor>s, and one or more calls to Ada/SQL DML statements. The program environment for an Ada/SQL system includes all <schema>s that may be accessed through it.

4.9. Procedures

Procedures, as defined by SQL, are not used in Ada/SQL.

4.10. Parameters

Parameters, as defined by SQL, are not used in Ada/SQL.

4.10.1. SQLCODE

The SQLCODE is a special program variable of type integer. Its value is set to a status code that either indicates that a call to the DML completed successfully or that an exception condition occurred during execution of the DML statement. *Note: Level 1 Ada/SQL will not set SQLCODE but will raise exceptions.*

4.10.1.1. Exceptions

Status in Ada/SQL is returned by raising the appropriate exception on error. The following exceptions are defined and discussed throughout this manual: UNIQUE_ERROR, NULL_ERROR, NOT_FOUND_ERROR.

4.10.2. Indicator Variables

An indicator variable is a program variable, constant or literal of type INDICATOR_VARIABLE, which is an enumeration type with values of NULL_VALUE and NOT_NULL, that is specified after another parameter in a call to a DML procedure. Its primary use is to indicate whether the value that the preceding parameter assumes or supplies is a null value.

4.11. Standard Programming Languages

Ada is the only programming language that the Ada/SQL system is designed to work with.

4.12. Cursors

A cursor is specified by a <declare cursor>.

For each <declare cursor> in a program environment, a cursor is effectively created by the execution of a <declare cursor> and destroyed when the program defining it terminates.

A cursor is in either the open state or the closed state. The initial state of a cursor is the closed state. A cursor is placed in the open state by an <open statement> and returned to the closed state by a <close statement>, a <commit statement>, or a <rollback statement>.

A cursor in the open state designates a table, an ordering of the rows of that table, and a position relative to that ordering. If the <declare cursor> does not specify an <order by clause>, then the rows of the table have an order defined by the underlying DBMS. This order is subject to the reproducibility requirement within a transaction, but it may change between transactions. Any program relying on an order defined by the underlying DBMS is erroneous.

The position of a cursor in the open state is either before a certain row, on a certain row, or after the last row. If a cursor is on a row, then that row is the current row of the cursor. A cursor may be before the first row or after the last row even though the table is empty.

A <fetch statement> advances the position of an open cursor to the next row of the cursors ordering and retrieves the values of the columns of that row. An <update *statement:* positioned> updates the current row of the cursor. A <delete statement: positioned> deletes the current row of the cursor.

If a cursor is before a row and a new row is inserted at that position, then the effect, if any on the position of the cursor is defined by the underlying DBMS. Any program relying on that position is erroneous.

If a cursor is on a row or before a row and that row is deleted, than the cursor is positioned before the row that is immediately after the position of the deleted row. If such a row does not exist, then the position of the cursor is after the last row.

If an error occurs during the execution of an <Ada/SQL statement> that identifies an open cursor, then the effect, if any, on the position or state of that cursor is defined by the underlying DBMS. Any program relying on the position or state of that cursor is erroneous.

The table designated by an open cursor is either a temporary base table or a temporary viewed table. The determination of whether a given cursor designates a temporary base table or a temporary viewed table is defined by the underlying DBMS. Any program relying on this is erroneous.

Each row of a temporary viewed table is derived only when the cursor is positioned on that row.

A temporary base table is created when the cursor is opened and destroyed when the cursor is closed.

4.13. Statements

An <Ada SQL statement> specifies a database operation or a cursor operation. A <select statement> fetches values from a table. An <insert statement> inserts rows into a table. A <update statement: searched> or <update statement:

positioned> updates the values in rows of a table. A <delete statement: searched> or <delete statement: positioned> deletes rows of a table.

4.14. Embedded Syntax

An <embedded Ada/SQL host program> uses exact Ada syntax that may be compiled by any standard, validated Ada compiler. The embedded SQL syntax conforms to Ada syntax.

4.15. Privileges

A privilege authorizes certain actions (insert, delete, select, and update) to be performed on a specified table by a specified <authorization identifier>. *Privileges are not defined by the Level 1 Ada/SQL DDL. The creation of tables and the setting of privileges is done outside of the Ada/SQL environment by the DBA using the DDL as required by the underlying DBMS.*

An <authorization identifier> is specified for each <schema>.

The <authorization identifier> specified for a <schema> shall be different from the <authorization identifier> of any other <schema> in the same environment. The <authorization identifier> of a <schema> is the "owner" of all tables and views defined in that <schema>.

Tables and views are designated by <table name>s. A <table name> consists of an <authorization identifier> and an <identifier>. The <authorization identifier> identifies the <schema> in which the table or view designated by the <table name> was defined. Tables and views defined in different <schema>s can have the same <identifier>. A <schema> has a single <authorization identifier>. It may, however, be defined within one or more <schema package declaration>s. Each schema package declaration is an Ada package.

If a reference to a <table name> does not explicitly contain an <authorization identifier>, then the appropriate <authorization identifier> of the containing <schema> is specified by default providing that reference was made within a <schema>. If the reference was not made within a <schema>, the applicable <schema package declaration> and corresponding <authorization identifier> are selected according to Ada visibility rules for table names without an explicitly stated authorization identifier or package name prefix.

The <authorization identifier> of a <schema> has all privileges on the tables and views defined in that <schema>.

A <schema> with a given <authorization identifier> may contain <privilege definition>s that grant privileges to other <authorization identifier>s. The granted privileges may apply to tables and views defined in the current <schema>, or they may be privileges that were granted to the given <authorization identifier> by other <schema>s. The WITH GRANT OPTION clause of a <privilege definition> specifies whether the recipient of a privilege may grant it to others. *<privilege definition>s are not implemented in Level 1 of Ada/SQL. All <privilege definition>s must be set through the underlying DBMS.*

4.16. Transactions

A transaction is a sequence of operations, including database operations, that is atomic with respect to recovery and concurrency. Transactions terminate with a <commit statement> or a <rollback statement>. If a transaction terminates with a <commit statement>, then all changes made to the database by that transaction are made accessible to all concurrent transactions. If a transaction terminates with a <rollback statement>, then all changes made to the database by that transaction are canceled. Committed changes cannot be canceled. Changes made to the database by a transaction can be perceived by that transaction, but cannot be perceived by other transactions until that transaction terminates with a <commit statement>.

The execution of concurrent transactions is guaranteed to be serializable. A serializable execution is defined to be an execution of the operations of concurrently executing transactions that produces the same effect as some serial execution of those same transactions. A serial execution is one in which each transaction executes to completion before the next transaction begins.

The execution of an <Ada/SQL statement> within a transaction has no effect on the database other than the effect stated in the General Rules for that <Ada/SQL statement>. Together with serializable execution, this implies that all read operations are reproducible within a transaction, except for changes explicitly made by the transaction itself.

Valid execution of any Ada/SQL data manipulation statement other than <declare cursor> initiates a transaction for the executing program, if one is not already in progress. A transaction in progress upon program termination is automatically terminated as if a <rollback statement> had been issued.

4.17. *Philosophy*

The limitations and constraints placed on the Ada statements described in this manual reflect the limitations of Level 1 Ada/SQL. For Level 1 of Ada/SQL we have elected to keep the DDL as similar to SQL's as possible without losing the advantage of Ada's strong typing and enumeration types. With this simplification of Ada/SQL to SQL format, we have lost some of the advanced features of Ada. These features will be implemented in a later Level of Ada/SQL. These restrictions of Level 1 affect only the part of an Ada program immediately concerned with Ada/SQL, the DDL and <Ada/SQL statement>s.

Level 1 Ada/SQL will not contain a DDL generator to generate DDL for the underlying Database Management System (DBMS)). The DBMS DDL must be created separately by the DBA. The DBMS DDL will contain table creation, setting unique columns, view creation, and privileges. Level 1 Ada/SQL DDL will be used only to describe table formats and column data types to the Ada/SQL system. A table must be defined to the database before the Ada/SQL System can access it. The classification package of the schema, as defined by the proposed Ada/SQL MIL-STD (but not required by the ANSI standard) which defines the security classification of all columns in each table defined in the schema packages, will not be implemented in Level 1. All security will be through privileges defined by the DBMS DDL.

Level 1 Ada/SQL DDL will be used only to describe table formats, column data types, range restrictions, and not null and unique limitations. Therefore the DDL package ADA_SQL will contain only type definitions. It may contain no function specifications, call no subprograms and declare no objects. Neither the ADA_SQL package nor its enclosing package is to have a body, since view definitions, privilege definitions and multiple column uniqueness constraints are not processed in Level 1 Ada/SQL DDL; these constructs will be placed in package bodies in later levels of Ada/SQL.

As mentioned earlier for Level 1 Ada/SQL we have elected to keep the DDL and DML as similar to SQL's as possible without losing the advantage of Ada's strong typing and enumeration types. Therefore we have eliminated some of the more advanced features of the type definitions for this level.

Following is a list of Ada features not allowed in the Level 1 Ada/SQL DDL.

1) Nested packages are not permitted except for the special case of the nested package ADA_SQL required in all DDL units.

2) Private sections are not permitted.

3) Package bodies are not permitted.

4) Ada attributes are not permitted.

5) Renaming declarations, generic declarations, generic instantiations, deferred constant declarations, subprogram declarations, task declarations, exception declarations, object declarations, and number declarations are not permitted.

6) Variables, constants and named numbers are not permitted.

7) All ranges, index constraints etc. are to be defined with literals, not with variables, constants or complex expressions.

8) All expressions are to be simple literals, no math may be performed, no functions such as ABS or NOT may be referenced, no relational operators such as = > < may be used and no variables or constants may be used.

9) Based literals are not permitted, all numeric literals must be decimal.

10) No default values may be assigned to any types.

11) Access types, private types, task types, incomplete type declarations and representation clauses may not be used.

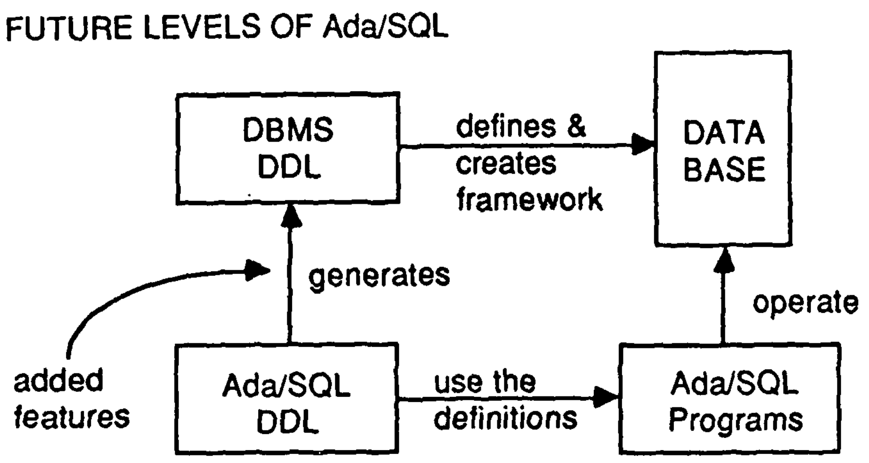
12) Array types must be made up of CHARACTER components and be of one dimension only with integer index.

13) Records may not contain discriminants, variant parts or unconstrained arrays; records must be defined as being of fixed length. Record components may not be of record types.

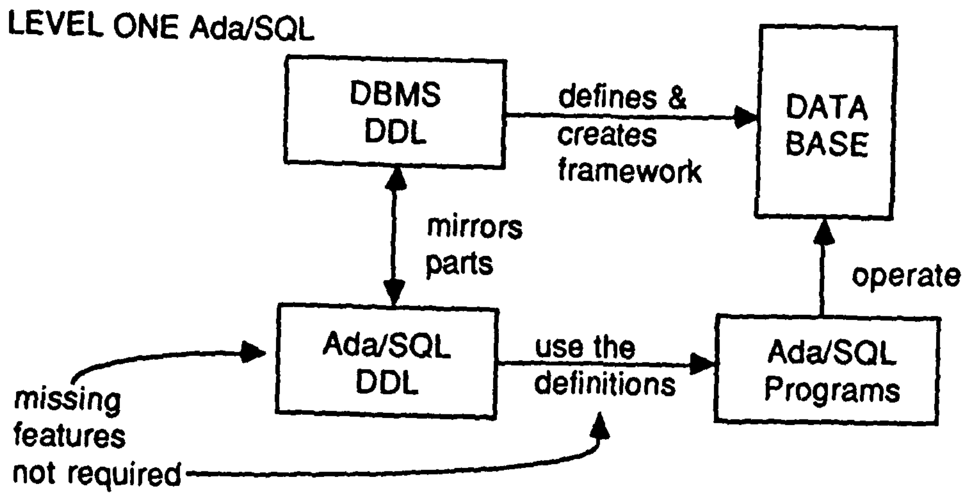
14) Fixed point numbers may not be used. Floating point and integer may be used.

15) Type conversions may not be used.

4.18. *Level 1 vs Future Levels*



This illustration shows the interaction of the different parts of Ada/SQL in Level 1. The DBMS DDL is used to define and create the framework of the database. The Ada/SQL DDL mirrors parts of the DBMS DDL, such as the format of tables. The definitions in the Ada/SQL DDL are used by the Ada/SQL programs to operate on the database. The features eliminated in Level 1 of Ada/SQL DDL are those not required by the Ada/SQL programs to operate on the database.



This illustration shows the interaction of the different parts of Ada/SQL in future levels. The Ada/SQL DDL is used to generate the DBMS DDL which is then used to define and create the framework of the database. The definitions in the Ada/SQL DDL are used by the Ada/SQL programs to operate on the database. The features added to future levels of Ada/SQL allow the DBMS DDL to be generated directly from the Ada/SQL DDL.

5. Common Elements

5.1. <character>

Function

Define the terminal symbols of the language and the elements of strings.

Format

```
<character> ::=
   <digit> | <letter> | <special character> | <space character>
<digit> ::=
   0 1 2 3 4 5 6 7 8 9

<letter> ::=
   <upper case letter> | <lower case letter>

<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

<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

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

<space character> ::=
  space

<format effector> ::=
  horizontal tabulation, vertical tabulation, carriage return,
  line feed, form feed
```

Syntax Rules

1) The only characters allowed in Ada/SQL are the <character>s defined here, including the format effectors.

General Rules

None.

5.2. <literal>

Function

Specify a nonnull value.

Format

```
<literal> ::=
      <enumeration literal>
    | <character string literal>
    | <numeric literal>

<enumeration literal> ::=
      <identifier>
    | <character literal>

<character literal> ::=
    '<character>'

<character string literal> ::=
   "<character representation>..."

<character representation> ::=
      <nonquote character>
    | <quote representation>

<nonquote character> ::=
   See Syntax Rule 4.

<quote representation> ::=
   ""

<numeric literal> ::=
      <decimal literal>

<decimal literal> ::=
    <integer> [.<integer>] [<exponent>]

<integer> ::=
    <digit> [{[<underscore>] <digit>}...]

<underscore> ::= _

<exponent> ::=
      E [+] <integer>
    | E - <integer>
```

Syntax Rules

1) An <enumeration literal> yields a value that belongs to an enumeration type.

2) A <character literal> is formed by enclosing one of the 95 <character>s (including the <space character>) between two apostrophe <character>s ('). A <character literal> has a value that belongs to a character type. The predefined type CHARACTER is an enumeration type with the values of the 128 characters of the ASCII code.

3) A <character string literal> is formed by a sequence of <character>s (possibly none) enclosed between two quotation characters used as string brackets.

4) A <nonquote character> is any <character> other than the double quote mark character (").

5) A <character string literal> yields a value that belongs to a character string type (a one-dimensional array of CHARACTER with integer index). The length of a <character string literal> is the number of <character representation>s that it contains. Each <quote representation> in a <character string literal> represents a single quotation mark character in both the value and the length of the <character string literal>.

6) There are two classes of <numeric literal>s: real literals and integer literals. A real literal is a <numeric literal> that includes a point (.); an integer literal is a <numeric literal> without a point. Real literals are the literals of the type universal_real. Integer literals are the literals of the type universal_integer. An exponent for an integer literal must not have a minus sign. Numeric literals may be automatically converted to any other similar (integer vs. real) numeric type.

7) An <underscore> character inserted between adjacent <digit>s of a <decimal literal> does not affect the value of this <numeric literal>. The letter E of the exponent, if any, can be written either in lower case or in upper case, with the same meaning.

8) A <numeric literal> may be a <based literal> in Ada. *In Level 1 Ada/SQL <based literal>s are not permitted. All numeric literals are assumed to be decimal.*

General Rules

1) The evaluation of a <literal> yields the corresponding value.

2) <enumeration literal>s include <character literal>s and yield values of the corresponding enumeration types.

3) A <character string literal> is a basic operation that combines the sequence of characters it contains into a value of a character string type; the lower bound of this array is given by S'FIRST where S is the index subtype; the upper bound is determined by the length.

4) For a <character string literal> containing no <character representation>s, the upper bound is the predecessor, as given by the PRED attribute, of the lower bound. The evaluation of a such a <character string literal> returns an error if the lower bound does not have a predecessor.

5) For the evaluation of a <character string literal> containing one or more <character representation>s, a check is made that the index range defined belongs to the index subtype. The exception CONSTRAINT_ERROR is raised if any of these checks fails.

6) The type of a <character string literal> must be determinable solely from the context in which this literal appears, excluding the literal itself, but using the fact that a <character string literal> is a value of a one dimensional array type whose component type is a character type.

7) The character literals corresponding to the <character representations> used within a <character string literal> must be visible at the place of the <character string literal> (although these characters themselves are not used to determine the type of the <character string literal>).

8) The numeric value of a <decimal literal> is derived from the normal mathematical interpretation of positional decimal notation (that is, the base is implicitly ten). An <exponent> indicates the power of ten by which the value of the <decimal literal> without the exponent is to be multiplied to obtain the value of the <decimal literal> with the <exponent>.

5.3. <token>

Function

Specify lexical units.

Format

```
<token> ::=
   <nondelimiter token> | <delimiter token>

<nondelimiter token> ::=
   <identifier>
 | <SQL key word>
 | <Ada key word>
 | <Ada/SQL key word>
 | <literal>

<SQL key word> ::=
   ALL | AND | ANY | AS | ASC | AUTHORIZATION | AVG | BEGIN | BETWEEN
 | BY | CHAR | CHARACTER | CHECK | CLOSE | COBOL | COMMIT | CONTINUE
 | COUNT | CREATE | CURRENT | CURSOR | DEC | DECIMAL | DECLARE | DELETE
 | DESC | DISTINCT | DOUBLE | END | ESCAPE | EXEC | EXISTS | FETCH
 | FLOAT | FOR | FORTRAN | FOUND | FROM | GO | GOTO | GRANT | GROUP
 | HAVING | IN | INDICATOR | INSERT | INT | INTEGER | INTO | IS
 | LANGUAGE | LIKE | MIX | MIN | MODULE | NOT | NULL | NUMERIC | OF
 | ON | OPEN | OPTION | OR | ORDER | PASCAL | PLI | PRECISION | PRIVILEGES
 | PROCEDURE | PUBLIC | REAL | ROLLBACK | SCHEMA | SECTION | SELECT | SET
 | SMALLINT | SOME | SQL | SQLCODE | SQLERROR | SUM | TABLE | TO | UNION
 | UNIQUE | UPDATE | USER | VALUES | VIEW | WHENEVER | WHERE | WITH | WORK

<Ada key word> ::=
   ABORT | ABS | ACCEPT | ACCESS | ALL | AND | ARRAY | AT | BEGIN | BODY
 | CASE | CONSTANT | DECLARE | DELAY | DELTA | DIGITS | DO | ELSE | ELSIF
 | END | ENTRY | EXCEPTION | EXIT | FOR | FUNCTION | GENERIC | GOTO | IF
 | IN | IS | LIMITED | LOOP | MOD | NEW | NOT | NULL | OF | OR | OTHERS
 | OUT | PACKAGE | PRAGMA | PRIVATE | PROCEDURE | RAISE | RANGE | RECORD
 | REM | RENAMES | RETURN | REVERSE | SELECT | SEPARATE | SUBTYPE | TASK
 | TERMINATE | THEN | TYPE | USE | WHEN | WHILE | WITH | XOR

<Ada/SQL key word> ::=
   ACS | ALLL | AND | ANY | AVG | AVG_ALL | AVG_DISTINCT | BETWEEN
 | CLOSE | COMMIT_WORK | COUNT | COUNT_DISTINCT | CURSOR_FOR | DECLAR
 | DELETE_FROM | DESC | EQ | ESCAPE | EXISTS | FETCH | FROM | GROUP_BY
 | HAVING | INDICATOR | INSERT_INTO | INTO | IS_IN | IS_NOT_NULL | IS_NULL
 | LIKE | MAX | MAX_ALL | MAX_DISTINCT | MIN | MIN_ALL | MIN_DISTINCT
 | NE | NOT | NOT_IN | NULL_VALUE | OPEN | OR | ORDER_BY | ROLLBACK_WORK
 | SELEC | SELECT_ALL | SELECT_DISTINCT | SET | SOME | SUM | SUM_ALL
 | SUM_DISTINCT | UNION | UNION_ALL | UPDATE | USER | VALUES | WHERE
 | WHERE_CURRENT_OF

<delimiter token> ::=
   & | ' | ( | ) | * | + | , | - | . | / | : | ; | < | = | > | {|} | =>
 | .. | ** | := | /= | >= | <= | << | >> | <>

<separator> ::=
   <space character>
 | <format effector>
```

| end of a line

<comment> ::=
  --[<character>...]<newline>

<newline> ::=
  implementor defined end of line indicator

Syntax Rules

1) A <token>, other than a <character string literal> or a <character literal>, shall not include a <space character>.

2) Any <token> may be followed by a <separator>. A <nondelimiter token> shall be followed by a <delimiter token> or a <separator>. If the syntax does not allow a <nondelimiter token> to be followed by a <delimiter token>, then that <nondelimiter token> shall be followed by a <separator>.

General Rules

None.

5.4. Names

Function

Specify names.

Format

```
<name> ::=
    <simple name>
  | <character literal>

<simple name> ::=
    <identifier>

<column name> ::=
    <identifier>

<table name> ::=
    <selected component table name>
  | <underscored table name>
  | <hyphenated table name>

<package name> ::=
    <unit simple name>
  | <unit simple name> [ . <unit simple name> ] ...

<unit simple name> ::=
    <identifier>

<selected component table name> ::=
  [ <authorization identifier> . ] <table identifier>

<underscored table name> ::=
  [ <authorization identifier> _ ] <table identifier>

<hyphenated table name> ::=
  [ <authorization identifier> - ] <table identifier>

<authorization identifier> ::=
    <identifier>

<table identifier> ::=
    <identifier>

<type identifier> ::=
    <identifier>

<correlation name declaration> ::=
  package <correlation name> is
   new <table name>_CORRELATION.NAME [ ("<identifier>") ] ;

<correlation name> ::=
    <identifier>

<subtype identifier> ::=
    <identifier> [ _NOT_NULL [ _UNIQUE ] ]
```

Syntax Rules

1) A <table name> identifies a named table. In most contexts where an <authorization identifier> is used within a <table name>, the <authorization identifier> is separated from the <table identifier> by a period. There are, however, isolated occurrences where the separator character is an underscore, (see the "<from clause>"), or a hyphen (see the "<view definition>" and the "<insert statement>". *<view definition> will be defined in a later Ada/SQL Level.*

2) If a <table name> does not contain an <authorization identifier>, then:

   a) If a <table name> is contained in a <schema>, then the <authorization identifier> specified as the <schema authorization identifier> of the <schema> is implicit.

   b) If the <table name> is contained outside a <schema>, then the <authorization identifier> as chosen by the Ada visibility rules is implicit.

3) Two <table name>s are equal if and only if they have the same <table identifier> and the same <authorization identifier>, regardless of whether the <authorization identifier>s are implicit or explicit.

4) A <table name> is declared by the <table definition> directly containing it.

5) A <table name> in an <Ada/SQL statement> shall identify a table defined in one of the <schema>s of the containing programs.

6) Case:

   a) If a <table name> is directly contained in a <correlation name declaration>, then it shall be written as an <underscored table name>.

   b) If a <table name> is directly contained in a <view definition> containing a <view column list> or an <insert statement> containing an <insert column list>, then it shall be written as a <hyphenated table name>. *<view definition> will be defined in a later Ada/SQL Level.*

   c) Otherwise, a <table name> shall be written as a <selected component table name>.

7) A <package name> *identifies* a *named* Ada package, which may be qualified to identify a nested package.

8) A <unit simple name> identifier an unqualified named Ada package.

9) An <authorization identifier> represents an authorization identifier.

10) The <identifier> of a <type identifier> shall have been defined previously with a type or subtype declaration.

11) An <identifier> is declared as a <correlation name> for a particular table by a <correlation name declaration> directly containing that <correlation name> and the <table name> that identifies the table. The same <correlation name> may be reused within different scopes of the same statement, although it must refer to different instances of the same table.

12) If the <correlation name declaration> contains the optional <identifier>, then that <identifier> shall match the <correlation name> declared.

13) The use of a <correlation name> in a <from clause> associates a scope with that use of the <correlation name>. The scope is either a <select statement>, <subquery>, or <query specification>. Scopes may be nested. The same <correlation name> may be associated with several different scopes, but may not be associated with the same scope more than once.

14) The <correlation name declaration> for a <correlation name> used in a <from clause> shall be directly visible, by Ada visibility rules, at the point in the source text at which the <from clause> appears.

15) A <column name> identifies a named column. An <identifier> is defined as a <column> by a <table definition>.

16) Names, other than authorization identifiers, table identifiers and column names defined in the DDL, may be any valid Ada identifiers, except that they may not be the same as <SQL key words>, <Ada key words> or <Ada/SQL key words>. Names of authorization identifiers, table identifiers and column

names are limited to 18 characters.

17) <subtype identifier>s may include the suffixes _NOT_NULL or _NOT_NULL_UNIQUE. Database columns defined by record components of subtypes named with these suffixes shall have the corresponding SQL constraints.

Level 1 Implementation Rules

1) *<authorization identifiers> are not used by UNIFY. UNIFY limits <table identifier>s to 8 characters and <column name>s to 16 characters.*

General Rules

None.

5.4.1. <identifier>

Function

Identifiers are used as names and reserved words.

Format

```
<identifier> ::=
    <letter> [{[<underscore>] <letter or digit>}...]

<underscore> ::= _

<letter or digit> ::=
    <letter>
  | <digit>
```

Syntax Rules

1) Identifiers perform their usual Ada functions within schemas, but are also used to name SQL authorization identifiers, tables, and columns.

2) An <identifier> may consist of as many characters as permitted by Ada, except <identifier>s relating directly to the database, such as authorization identifiers, table names, column names, etc, which may consist of no more than 18 <character>s.

3) A user-defined <identifier> shall not be identical to an <SQL key word>, an <Ada key word>, or an <Ada/SQL key word> as defined in Clause 5.3 "<token>".

4) Identifiers used as subtype names may include the suffixes _NOT_NULL or _NOT_NULL_UNIQUE. Database columns defined by record components of subtypes named with these suffixes shall have the corresponding SQL constraints.

5) Case is not significant in an <identifier>.

General Rules

None.

5.5. <data type>

Function

Specify a data type.

Format

```
<data type> ::=
    <string type definition>
  | <integer type definition>
  | <real type definition>
  | <enumeration type definition>
```

Syntax Rules

1) A database column of a subtype having a null range must not be declared as _NOT_NULL or _NOT_NULL_UNIQUE -- only null values may be stored within it. (The usefulness of such a column would be extremely limited.)

2) Ada/SQL supports the predefined type CHARACTER which is an enumeration type with the values of the 128 characters of the ASCII code.

3) Ada/SQL supports the predefined type BOOLEAN which is an enumeration type with the values of FALSE and TRUE.

4) Ada/SQL supports the predefined integer type of INTEGER, as well at the NATURAL (zero and greater than zero) and POSITIVE (greater than zero) subtypes. Other implementor optional predefined types defined in the STANDARD package, such as SHORT_INTEGER and LONG_INTEGER, are also supported, as appropriate for each implementation.

5) Ada/SQL supports the predefined real type FLOAT. Other implementor optional predefined types defined in the STANDARD package, such as SHORT_FLOAT and LONG_FLOAT, are also supported, as appropriate for each implementation.

6) Ada/SQL supports the predefined type STRING. The values of the predefined type STRING are one-dimensional arrays of the predefined type CHARACTER, indexed by values of the predefined subtype POSITIVE.

7) There may be other predefined types in the package STANDARD and/or other system dependent packages. All of these predefined types may be used in Ada/SQL, provided they are within a class described above.

General Rules

None.

5.5.1. <string type definition>

Function

Define a string data type.

Format

```
<string type definition> ::=
    <unconstrained array definition>
  | <constrained array definition>

<unconstrained array definition> ::=
    array (<index subtype definition>)
        of <component subtype indication>

<constrained array definition> ::=
    array <index constraint> of <component subtype indication>

<index subtype definition> ::=
    <type mark> range <>
```

Syntax Rules

1) For <string type definition> all arrays shall have a single integer index with components of CHARACTER type.

2) For <constrained array definition>, arrays shall have a single integer index with components of CHARACTER type; the <index constraint> shall have positive bounds; and the <component subtype indication> shall be of CHARACTER type and have no associated <constraint>.

3) For <unconstrained array definition>, the <type mark> of the <index subtype definition> shall be of an integer type. The <component subtype indication> shall be of CHARACTER type and have no associated <constraint>.

General Rules

None.

5.5.2. <integer type definition>

Function

Define an integer data type.

Format

```
<integer type definition> ::=
    <range constraint>
```

Syntax Rules

1) Each bound of the <range> of the <range constraint> shall be of an integer type.

General Rules

1) Integer values are positive or negative integers or zero. Integer types may have range constraints which identify a lower and upper limit of valid numbers associated with the type. The range of integers supported by a database management system may not be the same as that supported by the Ada system used to access that DBMS. The package DATABASE provides information about the DBMS being accessed by Ada/SQL. In particular, the smallest (most negative) integer value supported by the DBMS through Ada/SQL is the named number DATABASE.MIN_INT and the largest (most positive) value is DATABASE.MAX_INT. The type DATABASE.INTG is defined to encompass the maximum range of integers supported by the DBMS through Ada/SQL.

2) The DATABASE package also includes the definition of a type SMALLINT, with range corresponding to that supported by the DBMS type SMALLINT through Ada/SQL. Ada/SQL will convert Ada integer data types to the corresponding DBMS types as follows: (1) If the Ada type or subtype declaration explicitly references (following a chain of references) INTEGER or DATABASE.INTG, then the SQL INTEGER type is used, (2) if the declaration of the Ada type or subtype explicitly references SMALLINT, then the SQL SMALLINT type is used, (3) if none of these types is referenced in the Ada declarations, then the SQL SMALLINT type is used if the range of values is compatible with it, otherwise the SQL INTEGER type is used.

3) If the range of integers supported by the DBMS is smaller than that supported by Ada, then Ada/SQL will issue warning diagnostics upon encountering explicitly declared ranges that extend beyond the capability of the DBMS. The execution of an Ada/SQL operation that would require the DBMS to handle an integer beyond its range will cause an error indication to be returned to the program, unless the erroneous operation is performed entirely within the DBMS and is not detected by the DBMS.

4) SQL does not support subtypes, so database operations may be performed without range checking. (An implementation may perform range checking where practical, however, returning an error on database operations that would violate subtype constraints.) If range checking is not performed, it is possible for an Ada/SQL statement to cause one or more database columns to contain values outside the ranges defined for those columns. An error will be returned, however, when it is attempted to retrieve such values. If the value can be legally stored in the variable used to retrieve it, then the value will be stored before the error is returned.

5.5.3. <real type definition>

Function

Define a real data type.

Format

<real type definition> ::=
    <floating point constraint>

Syntax Rules

None.

General Rules

1) Real types represent approximations of numbers, with precision to a specified number of significant digits, and an optional restriction on their range. The range and accuracy of real numbers supported by a database management system may not be the same as that supported by the Ada system used to access that DBMS. If the range or accuracy of real numbers supported by the DBMS is smaller than that supported by Ada, then Ada/SQL will issue warning diagnostics upon encountering explicitly declared characteristics that extend beyond the capability of the DBMS. An error is returned by the execution of an Ada/SQL operation that would require the DBMS to handle a real number beyond its range, unless the erroneous operation is performed entirely within the DBMS and is not detected by the DBMS. In general, no error is returned if accuracy is lost as a result of database operations.

2) The underlying DBMS must support the model numbers (according to the Ada definition) for types that are successfully processed by Ada/SQL as well as safe numbers within the ranges of subtypes. The DBMS may also support a wider range of safe numbers.

3) The comments on range checking and errors for integers are applicable to real numbers as well.

4) The DATABASE package defines REAL and DOUBLE_PRECISION types, with ranges and accuracies corresponding to those supported by the SQL REAL and DOUBLE PRECISION types as available from the underlying DBMS through Ada/SQL. Ada/SQL will convert Ada floating point types to the corresponding DBMS types as follows: (1) If the Ada type or subtype declaration explicitly references (following a chain of references) DOUBLE_PRECISION, then the SQL DOUBLE PRECISION type is used, (2) if the declaration of the Ada type or subtype explicitly references REAL, then the SQL REAL type is used, (3) if neither DOUBLE_PRECISION nor REAL is referenced in the Ada declarations, then a SQL FLOAT type with appropriate precision is used if the range and accuracy of values is compatible with it, otherwise the SQL DOUBLE_PRECISION type is used. Note that the range and accuracy of the Ada FLOAT type may not correspond to those achievable with the SQL FLOAT type.

5) The maximum number of floating point digits that can be handled by Ada/SQL through the underlying DBMS is given by the system dependent named number DATABASE.MAX_DIGITS.

5.5.4. <enumeration type definition>

Function

Defines an enumeration data type.

Format

```
<enumeration type definition> ::=
    (<enumeration literal specification>
    {, <enumeration literal specification> } ... )

<enumeration literal specification> ::=
    <enumeration literal>
```

Syntax Rules

1) An <enumeration literal> shall be an <identifier> or <character literal>.

2) The <identifier>s and <character literal>s listed by an <enumeration type definition> must be distinct.

3) Each <enumeration literal> of an enumeration type has a position number which is an integer value. Each <enumeration literal> yields a different enumeration value. The predefined order relations between enumeration values follow the order of corresponding position numbers. The position number of the value of the first listed <enumeration literal> is zero; the position number for each other <enumeration literal> is one more than for its predecessor in the list.

General Rules

None.

5.5.5. <derived type definition>

Function

Define a derived data type.

Format

```
<derived type definition> ::=
    new <subtype indication>
```

Syntax Rules

1) A derived type definition defines a new (base) type whose characteristics are derived from those of a parent type; the new type is called a derived type. A derived type definition further defines a derived subtype, which is a subtype of the derived type.

2) The <subtype indication> defines the parent subtype. The parent type is the base type of the parent subtype. If a constraint exists for the parent subtype, a similar constraint exists for the derived subtype; the only difference is that for a range constraint and likewise for a floating constraint that includes a range constraint, the value of each bound is replaced by the corresponding value of the derived type.

3) The derived type belongs to the same class of types as the parent type. The set of possible values for the derived type is a copy of the set of possible values for the parent type.

4) Explicit type conversion of a value of the parent type into the corresponding value of the derived type is allowed and vice versa. (In Ada/SQL as in Ada, explicit type conversions may be performed from any numeric type to any numeric type and from any string type to any string type. *For Level 1 Ada/SQL, then, the additional explicit type conversion capability provided for derived types affects only enumeration types. Explicit type conversion from one enumeration type to another may be performed if one of the two types is derived from the other, directly or indirectly, or if there exists a third type from which both types are derived, directly or indirectly.*)

5) For each enumeration literal of the parent type (if applicable), there is a corresponding enumeration literal for the derived type.

General Rules

None.

5.5.6. <constraint>

Function

Define possible constraints for a data type.

Format

```
<constraint> ::=
     <range constraint>
   | <floating point constraint>
   | <index constraint>
```

Syntax Rules

1) A <range constraint> specifies the bounds of the values a type may contain.

2) A <floating point constraint> specifies the minimum number of significant decimal digits for a real type and possibly a <range constraint>.

3) An <index constraint> specifies the data type of the index of an array and the bounds of the index.

General Rules

None.

5.5.7. <range constraint>

Function

Define a range constraint for a data type.

Format

```
<range constraint> ::=
    range <range>

<range> ::=
    <range literal> .. <range literal>

<range literal> ::=
    [ + | - ] <numeric literal>
  | <enumeration literal>
```

Syntax Rules

1) A <range> specifies a subset of values of a scalar type. The <range> L..R specifies the values from L to R inclusive if the relation L <= R is true. The values L and R are called the lower bound and upper bound of the <range>, respectively. A value V is said to satisfy a <range constraint> if it belongs to the <range>; the value V is said to belong to the <range> if the relations L <= V and V <= R are both TRUE. A null <range> is a <range> for which the relation R < L is TRUE; no value belongs to a null <range>.

2) If a <range constraint> is used in a <subtype indication>, either directly or as part of a <floating point constraint>, the type of the <range literal> (converted from a universal type, if necessary) must be the same as the base type of the <type mark> of the <subtype indication>. A <range constraint> is compatible with a subtype if each bound of the <range> belongs to the subtype, or if the <range constraint> defines a null <range>; otherwise the <range constraint> is not compatible with the subtype. A <range constraint> on a subtype shall be a subset of the <range constraint> of the <subtype indicator>.

General Rules

None.

5.5.8. <floating point constraint>

Function

Define a floating point constraint for a real data type.

Format

```
<floating point constraint> ::=
    <floating accuracy definition> [<range constraint>]

<floating accuracy definition> ::=
    digits <numeric literal>
```

Syntax Rules

1) For <floating accuracy definition>, the <numeric literal> shall be a positive integer (nonzero).

2) The <numeric literal> specifies the minimum number of significant decimal digits that the representation of the real type being defined must provide.

3) Each bound of the <range> of the <range constraint> (if used) shall be of a real type.

General Rules

None.

5.5.9. <index constraint>

Function

Define an index constraint for an array data type.

Format

```
<index constraint> ::=
    (<range>)
  | (<subtype indication>)
```

Syntax Rules

1) The <index constraint> determines the range of possible values for the index of an array type, and thereby the corresponding array bounds.

2) The <index constraint> specifies the type of the array index, *which in Level 1 Ada/SQL must be an integer type*.

3) The <subtype indication> may only be the <type mark> of an integer type.

General Rules

None.

5.5.10. <subtype indication>

Function

Reference another (possibly constrained) type or subtype.

Format

```
<subtype indication> ::=
    <type mark> [ <constraint> ]

<type mark> ::=
    [ <package name> . ] <type identifier>
```

Syntax Rules

1) The <type mark> must previously have been defined with a <type declaration> or a <subtype declaration>.

2) Standard Ada visibility rules apply to the use or omission of the <package name> within the <type mark>.

General Rules

None.

5.6. <value specification>

Function

Indicate program values, indicate whether or not the values are null, implement the keyword USER.

Format

```
<value specification> ::=
    <program value>
  | <program value with indicator>

<program value> ::=
    <Ada program object>
  | <literal>
  | USER

<program value with indicator> ::=
    INDICATOR ( <program value> [ , <indicator variable> ] )
```

Syntax Rules

1) A <value specification> specifies a value that is not selected from a table.

2) An <indicator variable> is a program variable, constant, or literal of type INDICATOR_VARIABLE, which is an enumeration type with values of NULL_VALUE and NOT_NULL.

3) An <Ada program object> is a program variable defined within a <variable package>, of a type appropriate for the database column being accessed.

4) The data type of USER is USER_AUTHORIZATION_IDENTIFIER, defined in the DATABASE package.

5) The data type of a <program value with indicator> is the same as that of the contained <program value>.

Level 1 Implementation Rules

1) *Null values shall not be used with UNIFY.*

General Rules

1) If a <value specification> contains an <indicator variable> and the value of the <indicator variable> is NULL_VALUE, then the value specified by the <value specification> is null. Otherwise, the value specified by a <value specification> is the value of the contained <program value>.

2) The value specified by a <literal> is the value represented by that <literal>.

3) The value specified by USER is indicative of the user executing the program.

5.7. <column specification>

Function

Reference a named column.

Format

```
<column specification> ::=
    [ <qualifier>. ] <column name>

<qualifier> ::=
    <table name>
  | <correlation name>
```

Syntax Rules

1) A <column specification> references a named column. The meaning of a reference to a column depends on the context.

2) Let C be the <column name> of the <column specification>.

3) Case:

   a) If a <column specification> contains a <qualifier>, then the <column specification> shall appear within the scope of one or more <table name>s or <correlation names>s equal to that <qualifier>. If there is more than one such <table name> or <correlation name>, then the one with the most local scope is specified. The table associated with the specified <table name> or <correlation name> shall include a column whose <column name> is C.

   b) If a <column specification> does not include a <qualifier>, then it shall be contained within the scope of one or more <table name>s or <correlation name>s. Of these, let the phrase "possible *qualifiers" denote those <table name>s and* <correlation name>s whose associated table includes a column whose <column name> is C. There shall be exactly one possible qualifier with the most local scope, and that <table name> or <qualifier> is implicitly specified.

   Note: the "scope" of a <table name> or <correlation name> is specified in 5.20, "<from clause>", 8.5, "<delete statement: searched>", and 8.12, "<update statement: searched>"

4) If a <column specification> is contained in a <table expression> T and the scope of the implicitly or explicitly specified <qualifier> of the <column specification> is some <Ada/SQL statement> or <table expression> that contains the <table expression> T, then the <column specification> is an "outer reference" to the table associated with that <qualifier>.

5) Let T denote the table associated with the explicitly or implicitly specified <qualifier> R. The data type of a <column specification> is the data type of column C of T.

General Rules

1) "C" or "R.C" references column C in a given row of T.

5.8. <set function specification>

Function

Specify a value derived by the application of a function to an argument.

Format

```
<set function specification> ::=
    COUNT ( '*' )
  | <distinct set function>
  | <all set function>

<distinct set function> ::=
    {  AVG_DISTINCT
  |  MAX_DISTINCT
  |  MIN_DISTINCT
  |  SUM_DISTINCT
  |  COUNT_DISTINCT }  ( <column specification> )

<all set function> ::=
    {  AVG
  |  MIX
  |  MIN
  |  SUM
  |  AVG_ALL
  |  MAX_ALL
  |  MIN_ALL
  |  SUM_ALL   }  ( <value expression> )
```

Syntax Rules

1) The argument of COUNT ('*') and the argument source of a <distinct set function> and <all set function> is a table or a group of a grouped table as specified in 5.19 "<table expression>", 5.24 "<subquery>" and 5.25 "<query specification>".

2) Let R denote the argument or argument source of a <set function specification>.

3) The <column specification> of a <distinct set function> and each <column specification> in the <value expression> of an <all set function> shall unambiguously reference a column of R and shall not reference a column derived from a <set function specification>.

4) The <value expression> of an <all set function> shall include a <column specification> that references a column of R and shall not include a <set function specification>. If the <column specification> is an outer reference, then the <value expression> shall not include any operators. Note: "outer reference" is defined in 5.7 "<column specification>".

5) The value returned by a set function, other than a count set function, is typed the same as the <column specification> or <value expression> argument of the set function. The value returned by a count set function is of type INTG defined in the DATABASE package.

6) If a <set function specification> contains a <column specification> that is an outer reference, then the <set function specification> shall be contained in a <subquery> of a <having clause>. Note: "outer reference" is defined in 5.7 "<column specification>".

7) Let T be the data type of the values that result from evaluation of the <column specification> or <value expression>.

8) If SUM or AVG is specified, then T shall not be a character string or an enumeration.

Level 1 Implementation Rules

1) <distinct set function>s shall not be used with UNIFY.

2) The <value expression> in an <all set function> shall not be of a string type with UNIFY.

General Rules

1) The argument of a <distinct set function> is a set of values. The set is derived by the elimination of any null values and any redundant duplicate values from the column of R referenced by the <column specification>.

2) The argument of an <all set function> is a multi-set of values. The multi-set is derived by the elimination of any null values from the result of the application of the <value expression> to each row of R. The specification or omission of _ALL does not affect the meaning of an <all set function>.

3) Let S denote the argument of a <distinct set function> or an <all set function>.

4) Case:

   a) If the <distinct set function> COUNT_DISTINCT is specified, then the result is the cardinality of S.

   b) If COUNT ('*') is specified, then the result is the cardinality of R.

   c) If AVG, MIX, MIN, or SUM (with or without _ALL or _DISTINCT suffix) is specified and S is empty, then the result is the null value.

   d) If MIX or MIN (with or without _ALL or _DISTINCT suffix) is specified, then the result is respectively the maximum or minimum value in S. These results are determined using the comparison rules specified in 5.11 "<comparison predicate>".

   e) If SUM (with or without _ALL or _DISTINCT suffix) is specified, then the result is the sum of the values in S. The sum shall be within the range of the data type of the result.

   f) If AVG (with or without _ALL or _DISTINCT suffix) is specified, then the result is the average of the values in S. The sum of the values in S shall be within the range of the data type of the result.

5.9. <value expression>

Function

Specify a (possibly) computed value.

Format

```
<value expression> ::=
   [ + | - ] <term>
 | <value expression> + <term>
 | <value expression> - <term>

<term> ::=
   <factor>
 | <term> * <factor>
 | <term> / <factor>

<factor> ::=
   <primary>

<primary> ::=
   <value specification>
 | <column specification>
 | <set function specification>
 | ( <value expression> )
 | <type conversion>
```

Syntax Rules

1) A <value expression> that includes a <distinct set function> shall not include any dyadic operators.

2) The first <character> of the <token> following a monadic operator shall not be a plus or minus sign.

3) If the data type of a <primary> is character string or enumeration , then the <value expression> shall not include any operators. The data type of the result is the same as that of the <primary>.

4) The data type of the result of a numeric operation is the same as that of the operand(s). Both operands to a dyadic operation shall be of the same type.

5) Note that standard SQL permits monadic "+" or "-" operators before any <primary> used within a <value expression>. The corresponding Ada unary_adding_operators may be applied only to an entire simple_expression. Furthermore, a leading Ada unary_adding_operator is applied to the entire first term within a simple_expression, while a leading SQL monadic operator in a similar <value expression> would be applied to the first <factor> within the <term>. Expressions written in Ada/SQL are interpreted according to Ada rules. Due to the nature of the operations, however, the arithmetic results will be the same as if SQL interpretation had been applied. Furthermore, any SQL <value expression> may be equivalently stated in Ada, using parentheses or depending on the properties of the arithmetic operators, even though the Ada syntax is more restrictive.

Level 1 Implementation Rules

1) *The +, -, *, and / operators for database types shall not be redefined.*

General Rules

1) If the value of any <primary> is the null value, then the result of the <value expression> is the null value.

2) If operators are not specified, then the result of the <value expression> is the value of the specified <primary>.

3) When a <value expression> is applied to a row of a table, each reference to a column of that table is a reference to the value of that column in that row.

4) The monadic arithmetic operators + and - specify monadic plus and monadic minus, respectively. Monadic plus does not change its operand. Monadic minus reverses the sign of its operand.

5) The dyadic arithmetic operators +, -, * and / specify addition, subtraction, multiplication, and division, respectively. A divisor shall not be 0.

6) The result of an operation applied to one or more database values shall be appropriately representable by the database rules.

7) Expressions within parentheses are evaluated first and when the order of evaluation is not specified by parentheses, multiplication and division are applied before any leading monadic operator, which is applied before addition and subtraction, and operators at the same precedence level are applied from left to right.

5.10. <predicate>

Function

Specify a condition that can be evaluated to give a truth value of "true", "false", or "unknown".

Format

```
<predicate> ::=
     <comparison predicate>
   | <between predicate>
   | <in predicate>
   | <like predicate>
   | <null predicate>
   | <quantified predicate>
   | <exists predicate>
```

Syntax Rules

None.

General Rules

1) The result of a <predicate> is derived by applying it to a given row of a table.

5.11. <comparison predicate>

Function

Specify a comparison of two values.

Format

```
<comparison predicate> ::=
    <equality operator> ( <value expression> , <right comparison operand> )
  | <value expression> <ordering operator> <right comparison operand>

<equality operator> ::=
    EQ | NE

<ordering operator> ::=
    < | > | <= | >=

<right comparison operand> ::=
    <value expression>
  | <subquery>
```

Syntax Rules

1) The <right comparison operand> of a <comparison predicate> may be either a (possibly computed) value or a <subquery>.

2) The data types of the first <value expression> and the <right comparison operand> shall be the same.

Level 1 Implementation Rules

1) *The <, >, <= and >= operators for database types shall not be redefined.*

General Rules

1) Let x denote the result of the first <value expression> and let y denote the result of the <right comparison operand>. The result of a <subquery> shall be at most one value.

2) If x or y is the null value or if the result of the <subquery> is empty, then "<equality operator> (x, y)" or "x <ordering operator> y" is unknown.

3) If x or y are nonnull values, then "<equality operator> (x, y)" or "x <ordering operator> y" is either true or false:

"EQ (x, y)" is true if and only if x and y are equal.
"NE (x, y)" is true if and only if x and y are not equal.
"x < y" is true if and only if x is less than y.
"x > y" is true if and only if x is greater than y.
"x <= y" is true if and only if x is not greater than y.
"x >= y" is true if and only if x is not less than y.

4) Numbers, integers, and reals are compared with respect to their algebraic value.

5) The comparison of two character strings is determined by the comparison of <character>s with the same ordinal position. If the strings do not have the same length, then the comparison is made with a temporary copy of the shorter string that has been effectively extended on the right with <space>s so that it has the same length as the other string.

6) Two strings are equal if all <character>s with the same ordinal position are equal. If two strings are not equal, then their relation is determined by the comparison of the first pair of unequal <character>s from the left end of the strings. This comparison is made with respect to the ASCII collating sequence.

7) Ordering and comparison of enumeration values follow the order of the corresponding position numbers

of the enumeration literals.

8) Although "EQ (x, y)" is unknown if both x and y are null values, in the contexts of GROUP BY, ORDER BY and DISTINCT, a null value is identical to or is a duplicate of another null value.

5.11.1. <type conversion>

Function

Specify the conversion of a value from one type to another.

Format

```
<type conversion> ::=
    CONVERT_TO.<type mark> ( <value expression> )
```

Syntax Rules

1) The type of the result is the same as the <type mark>.

2) Numerics may be converted to and from all other numerics. Strings may be converted to and from all other strings. Enumerations may be converted to and from other enumerations when one was derived from the other or when both were derived from a third enumeration.

3) The <value expression> shall contain at least on database <column name>.

4) The conversion of derived types is allowed if the <left operand> type is derived from the <right operand> type, directly or indirectly or vice versa.

5) The <type mark> shall be the fully expanded name of the result type of the conversion. This type shall be defined in one of the <schema package>s named in the <context clause>.

General Rules

1) For underlying DBMSs not supporting strong data typing, type conversions merely affect comparability and assignability of values, no type checking is really performed.

5.12. <between predicate>

Function

Specify a range comparison.

Format

```
<between predicate> ::=
   [ NOT ] BETWEEN ( <test value>,
        <value expression> AND <value expression> )

<test value> ::=
   <value expression>
```

Syntax Rules

1) The data types of the <test value> and the two <value expression>s shall be the same.

General Rules

1) Let x denote the <test value>, and y and z denote the first and second <value expression>s, respectively.

2) "BETWEEN (x, y AND Z)" has the same result as " x >= y AND x <= Z".

3) "NOT BETWEEN (x, y AND z)" has the same result as "NOT (x <= y AND x <= z)".

5.13. <in predicate>

Function

Specify a quantified comparison.

Format

```
<in predicate> ::=
    { IS_IN | NOT_IN }
     ( <value expression> , { <subquery> | <in value list> } )

<in value list> ::=
    <value specification> [ { or <value specification> } ... ]
```

Syntax Rules

1) The data types of the <value expression> and the <subquery> or all <value specification>s in the <in value list> shall be the same.

General Rules

1) Let X denote the result of the <value expression>. Let S denote the result of the <subquery> as in a <quantified predicate>, or the values specified by the <in value list>.

2) "IS_IN (X, S)" has the same result as "EQ (X, (ANY (S))". "NOT_IN (X, S)" has the same result as "NOT (IS_IN (X, S))".

5.14. <like predicate>

Function

Specify a pattern-match comparison.

Format

```
<like predicate> ::=
    [ NOT ] LIKE ( <column specification> , <pattern>
            [ , ESCAPE => <escape character> ] )

<pattern> ::=
    <value specification>

<escape character> ::=
    <value specification>
```

Syntax Rules

1) The <column specification> is the specification of the column to be tested. The <column specification> shall reference a character string column.

2) The data type of the <pattern> shall be the same as that of the <column specification>.

3) The <escape character> shall reference a character string with a length of one, of the same type as the pattern.

Level 1 Implementation Rules

1) *An <escape character> may not be specified with UNIFY.*

2) *The <pattern> may not include ? or * characters with UNIFY.*

General Rules

1) Let x denote the value referenced by the <column specification> and let y denote the result of the <value specification> of the <pattern>.

2) Case:

a) If an <escape character> is specified, then:

i) Let z denote the result of the <value specification> of the <escape character>.

ii) There shall be a partitioning of the string y into substrings such that each substring is of length 1 or 2, no substring of length 1 is the escape character z, and each substring of length 2 is the escape character z followed by either the escape character z, an underscore character, or the percent sign character. In that partitioning of y, each substring of length 2 represents a single occurrence of the second character of that substring. Each substring of length 1 that is the underscore character represents an arbitrary character specifier. Each substring of length 1 that is the percent sign character represents an arbitrary string specifier. Each substring of length 1 that is neither the underscore character nor the percent sign character represents the character that it contains.

b) If an <escape character> is not specified, then each underscore character in y represents an arbitrary character specifier, each percent sign character in y represents an arbitrary string specifier, and each character in y that is neither the underscore character nor the percent sign character represents itself.

3) The string y is a sequence of the minimum number of substring specifiers such that each <character> of y is part of exactly one substring specifier. A substring specifier is an arbitrary character specifier, an arbitrary string specifier, or any sequence of <character>s other than an arbitrary character specifier or

an arbitrary string specifier.

4) "LIKE (x, y)" is unknown if x or y is the null value. If x and y are nonnull values, then "LIKE (x, y)" is either true or false.

5) "LIKE (x, y)" is true if there exists a partitioning of x into substrings such that:

   a) A substring of x is a sequence of zero or more contiguous <character>s of x and each <character> of x is part of exactly one substring.

   b) If the nth substring specifier of y is an arbitrary character specifier, the nth substring of x is any single <character>.

   c) If the nth substring specifier of y is an arbitrary string specifier, the nth substring of x is any sequence of zero or more <character>s.

   d) If the nth substring specifier of y is neither an arbitrary character specifier nor an arbitrary string specifier, the nth substring of x is equal to that substring specifier and has the same length as that substring specifier.

   e) The number of substrings of x is equal to the number of substring specifiers of y.

6) "NOT LIKE (x, y)" has the same result as "NOT (LIKE (X, y))".

5.15. <null predicate>

Function

Specify a test for a null value.

Format

```
<null predicate> ::=
    { IS_NULL | IS_NOT_NULL } ( <column specification> )
```

Syntax Rules

None.

Level 1 Implementation Rules

1) *<null predicate>s shall not be used with UNIFY.*

General Rules

1) Let x denote the value referenced by the <column specification>.

2) "IS_NULL (x)" is either true or false.

3) "IS_NULL (x)" is true if and only if x is the null value.

4) "IS_NOT_NULL (x)" has the same result as "NOT IS_NULL (x)".

5.16. <quantified predicate>

Function

Specify a quantified comparison.

Format

```
<quantified predicate> ::=
    <equality operator> ( <value expression> , <quantified subquery> )
  | <value expression> <ordering operator> <quantified subquery>

<quantified subquery> ::=
    <quantifier> ( <subquery> )

<quantifier> ::=
    <all>
  | <some>

<all> ::=
    ALLL

<some> ::=
    SOME
  | ANY
```

Syntax Rules

1) The data types of the <value expression> and the <quantified subquery> shall be the same.

Level 1 Implementation Rules

1) *<qualified predicate>s shall not be used with UNIFY.*

General Rules

1) Let X denote the result of the <value expression> and let S denote the result of the <subquery>.

2) The result of "<equality operator> (X, <quantifier> S)" or "X <ordering operator> <quantifier> S" is derived by the application of the implied <comparison predicate> "<equality operator> (X, S)" or "X <ordering operator> S" to every value in S.

3) Case:

a) If S is empty or if the implied <comparison predicate> is true for every value s in S, then "<equality operator> (X, <all> S)" and "X <ordering operator> <all> S" are true.

b) If the implied <comparison predicate> is false for at least one value s in S, then "<equality operator> (X, <all> S)" and "X <ordering operator> <all> S" are false.

c) If the implied <comparison predicate> is true for at least one value s in S, then "<equality operator> (X, <some> S)" and "X <ordering operator> <some> S" are true.

d) If S is empty or if the implied <comparison predicate> is false for every value s in S, then "<equality operator> (X, <some> S)" and "X <ordering operator> <some> S" are false.

e) If "<equality operator> (X, <quantifier> S)" or "X <ordering operator> <quantifier> S" is neither true nor false, then it is unknown.

5.17. <exists predicate>

Function

Specify a test for an empty set.

Format

```
<exists predicate> ::=
    EXISTS ( <subquery> )
```

Syntax Rules

None.

Level 1 Implementation Rules

1) *<exists predicate>s shall not be used with UNIFY.*

General Rules

1) Let S denote the result of the <subquery>.

2) "EXISTS (S)" is either true or false.

3) "EXISTS (S)" is true if and only if S is not empty.

5.18. <search condition>

Function

Specify a condition that is "true", "false", or "unknown" depending on the result of applying boolean operators to specified conditions.

Format

```
<search condition> ::=
    <boolean factor> [ { AND <boolean factor> } ... ]
  | <boolean factor> [ { OR <boolean factor> } ... ]

<boolean factor> ::=
    [ NOT ] <boolean primary>

<boolean primary> ::=
    <predicate>
  | ( <search condition> )
```

Syntax Rules

1) Combinations of ANDs and ORs shall be parenthesized to clearly show order of evaluation. <This differs from SQL syntax but is required by Ada syntax>.

2) A <column specification> or <value expression> specified in a <search condition> is directly contained in that <search condition> if the <column specification> or <value expression> is not specified within a <set function specification> or a <subquery> of the <search condition>.

General Rules

1) The result is derived by the application of the specified boolean operators to the conditions that result from the application of each specified <predicate> to a given row of a table or a given group of a grouped table. If boolean operators are not specified, then the result of the <search condition> is the result of the specified <predicate>.

2) NOT (true) is false, NOT (false) is true, and NOT (unknown) is unknown. AND and OR are defined by the following truth tables:

AND	True	False	Unknown
True	True	False	Unknown
False	False	False	False
Unknown	Unknown	False	Unknown

OR	True	False	Unknown
True	True	True	True
False	True	False	Unknown
Unknown	True	Unknown	Unknown

5) Expressions within parentheses are evaluated first. When used without parentheses, NOT is applied

before AND or OR. Unparenthesized operators of the same type are applied from left to right.

6) When a <search condition> is applied to a row of a table, each reference to a column of that table by a <column specification> directly contained in the <search condition> is a reference to the value of that column in that row.

5.19. <table expression>

Function

Specify a table or a grouped table.

Format

```
<table expression> ::=
    <from clause>
  [ , <where clause> ]
  [ , <group by clause> ]
  [ , <having clause> ]
```

Syntax Rules

1) If the table identified in the <from clause> is a grouped view, then the <table expression> shall not contain a <where clause>, <group by clause>, or <having clause>.

General Rules

1) If all optional clauses are omitted, then the table is the result of the <from clause>. Otherwise, each specified clause is applied to the result of the previously specified clause and the table is the result of the application of the last specified clause. The result of a <table expression> is a derived table in which the n^{th} column inherits the description of the n^{th} column of the table specified by the <from clause>.

5.20. <from clause>

Function

Specify a table derived from one or more named tables.

Format

```
<from clause> ::=
   FROM => <table reference> [ { & <table reference> } ... ]

<table reference> ::=
   [ <correlation name> . ] <table name>
```

Syntax Rules

1) A <table name> specified in a <table reference> is exposed in the containing <from clause> if and only if that <table reference> does not specify a <correlation name>.

2) A <table name> that is exposed in a <from clause> shall not be the same as any other <table name> that is exposed in that <from clause>.

3) A <correlation name> specified in a <table reference> shall not be the same as any other <correlation name> specified in the containing <from clause>, and shall not be the same as the <table identifier> of any <table name> that is exposed in the containing <from clause>.

4) The scope of <correlation name>s and exposed <table name>s specified in a <from clause> is the innermost <subquery>, <query specification>, or <select statement> that contains the <table expression> in which the <from clause> is contained. A <table name> that is specified in a <from clause> has a scope defined by that <from clause> if and only if the <table name> is exposed in that <from clause>.

5) *If the table identified by <table name>* is a grouped view, then the <from clause> shall contain exactly one <table reference>.

6) Case:

   a) If the <from clause> contains a single <table name>, then the description of the result of the <from clause> is the same as the description of the table identified by that <table name>.

   b) If the <from clause> contains more than one <table name>, then the description of the result of the <from clause> is the concatenation of the descriptions of the tables identified by those <table name>s, in the order in which the <table name>s appear in the <from clause>.

7) In Ada/SQL, a <correlation name> is actually a package instantiated from the generic package specific to each <table name>. In order to define a <correlation name>, the appropriate generic package must be instantiated, in one of the two following ways:

```
package <correlation name> is new
   <table identifier>_CORRELATION_NAME;

package <correlation name> is new
   <authorization identifier>_<table identifier>_CORRELATION_NAME;
```

   Note that the generic packages are, in general, named <table name>_CORRELATION_NAME, except that an <authorization identifier> used within a <table name> is separated from the <table identifier> by an underscore.

   Although <correlation name>s are specifically declared to pertain to specific tables, the same <correlation name> may be reused within different scopes of the same statement, to refer to different instances of the same table.

General Rules

1) The specification of a <correlation name> or exposed <table name> in a <table reference> defines that <correlation name> or <table name> as a designator of the table identified by the <table name> of that <table reference>.

2) Case:

   a) If the <from clause> contains a single <table name>, then the result of the <from clause> is the table identified by that <table name>.

   b) If the <from clause> contains more than one <table name>, then the result of the <from clause> is the extended Cartesian product of the tables identified by those <table names>s. The extended Cartesian product, R, is the multi-set of all rows r such that r is the concatenation of a row from each of the identified tables in the order in which they are identified. The cardinality of R is the product of the cardinalities of the identified tables. The ordinal position of a column in R is n+s, where n is the ordinal position of that column in the named table T from which it is derived and s is the sum of the degrees of the tables identified before T in the <from clause>.

5.21. <where clause>

Function

Specify a table derived by the application of a <search condition> to the result of the preceding <from clause>.

Format

```
<where clause> ::=
    WHERE => <search condition>
```

Syntax Rules

1) Let T denote the description of the result of the preceding <from clause>. Each <column specification> directly contained in the <search condition> shall unambiguously reference a column of T or be an outer reference. Note: "outer reference is defined in 5.7 "<column specification>".

2) A <value expression> directly contained in the <search condition> shall not include a reference to a column derived from a function.

3) If a <value expression> directly contained in the <search condition> is a <set function specification>, then the <where clause> shall be contained in a <having clause> and the <column specification> in the <set function specification> shall be an outer reference.

General Rules

1) Let R denote the result of the <from clause>.

2) The <search condition> is applied to each row of R. The result of the <where clause> is a table of those rows of R for which the result of the <search condition> is true.

3) Each <subquery> in the <search condition> is effectively executed for each row of R and the results used in the application of the <search condition> to the given row of R. If any executed <subquery> contains an outer reference to a column R, then the reference is to the value of that column in the given row of R.

5.22. <group by clause>

Function

Specify a grouped table derived by the application of the <group by clause> to the result of the previously specified clause.

Format

```
 <group by clause> ::=
    GROUP_BY => <column specification> [ { & <column specification> } ... ]
```

Syntax Rules

1) Let T denote the description of the result of the preceding <from clause> or <where clause>.

2) Each <column specification> in the <group by clause> shall unambiguously reference a column of T. A column referenced in a <group by clause> is a grouping column.

General Rules

1) Let R denote the result of the preceding <from clause> or <where clause>.

2) The result of the <group by clause> is a partitioning of R into a set of groups. The set is the minimum number of groups such that, for each grouping column of each group of more than one row, all values of that grouping column are identical.

3) Every row of a given group contains the same value of a given grouping column. When a <search condition> or <value expression> is applied to a group, a reference to a grouping column is a reference to that value.

5.23. <having clause>

Function

Specify a restriction on the grouped table resulting from the previous <group by clause> or <from clause> by eliminating groups not meeting the <search condition>.

Format

```
<having clause> ::=
    HAVING => <search condition>
```

Syntax Rules

1) Let T denote the description of the result of the preceding <from clause>, <where clause>, or <group by clause>. Each <column specification> directly contained in the <search condition> shall unambiguously reference a grouping column of T or be an outer reference. Note: "outer reference" is defined in 5.7 "<column specification>".

2) Each <column specification> contained in a <subquery> in the <search condition> that references a column of T shall reference a grouping column of T or shall be specified within a <set function specification>.

General Rules

1) Let R denote the result of the preceding <from clause>, <where clause>, or <group by clause>. If that clause is not a <group by clause>, the R consists of a single group and does not have a grouping column.

2) The <search condition> is applied to each group of R. The result of the <having clause> is a grouped table of those groups of R for which the result of the <search condition> is true.

3) When the <search condition> is applied to a given group of R, that group is the argument or argument source of each <set function specification> directly contained in the <search condition> unless the <column specification> in the <set function specification> is an outer reference.

4) Each <subquery> in the <search condition> is effectively executed for each group of R and the result used in the application of the <search condition> to the given group of R. If any executed <subquery> contains an outer reference to a column of R, then the reference is to the values of that column in the given group of R.

5.24. <subquery>

Function

Specify a multi-set of values derived from the result of a <table expression>.

Format

```
<subquery> ::=
   [ SELEC | SELECT_ALL | SELECT_DISTINCT ]
     ( <subquery result specification> , <table expression> )

<subquery result specification> ::=
    <value expression>
  | '*'
```

Syntax Rules

1) The applicable <privilege>s for each <table name> contained in the <table expression> shall include SELEC. *Note: the "applicable <privileges>" for a <table name> will be defined in "<privilege definition>" for later levels of Ada/SQL. In Level 1, privileges are as defined to the underlying DBMS.*

2) Case:

   a) If the <subquery result specification> '*' is specified in a <subquery> of any <predicate> other than an <exists predicate>, then the degree of the <table expression> shall be 1, and the <subquery result specification> is equivalent to a <value expression> consisting of a <column specification> that references the sole column of the <table expression>.

   b) If the <subquery result specification> "*" is specified in a <subquery> of an <exists predicate>, then the <subquery result specification> is equivalent to an arbitrary <value expression> that does not include a <set function specification> and that is allowed in the <subquery>.

3) The data type of the values of the <subquery> is the data type of the implicit or explicit <value expression>.

4) Let R denote the result of the <table expression>.

5) Each <column specification> in the <value expression> shall unambiguously reference a column of R.

6) If R is a grouped view, then the <subquery result specification> shall not contain a <set function specification>.

7) If R is a grouped table, then each <column specification> in the <value expression> shall reference a grouping column or be specified within a <set function specification>. If R is not a grouped table and the <value expression> includes a <set function specification>, then each <column specification> in the <value expression> shall be specified within a <set function specification>.

8) A <subquery>, excluding any <subquery> contained within it, shall contain at most one use of an Ada/SQL keyword ending in DISTINCT (SELECT_DISTINCT, AVG_DISTINCT, MAX_DISTINCT, MIN_DISTINCT, SUM_DISTINCT, COUNT_DISTINCT).

9) If a <subquery> is specified in a <comparison predicate>, then the <table expression> shall not contain a <group by clause> or a <having clause> and shall not identify a grouped view.

General Rules

1) If R is not a grouped table and the <value expression> includes a <set function specification>, then R is the argument or argument source of each <set function specification> in the <value expression> and the result of the <subquery> is the value specified by the <value expression>.

2) If R is not a grouped table and the <value expression> does not include a <set function specification>, then the <value expression> is applied to each row of R yielding a multi-set of n values, where n is the cardinality of R. If SELECT_DISTINCT is not specified, then the multi-set is the result of the

<subquery>. If SELECT_DISTINCT is specified, then the result of the <subquery> is the set of values derived from that multi-set by the elimination of all redundant duplicate values.

3) If R is a grouped table, then the <value expression> is applied to each group of R yielding a multi-set of n values, where n is the number of groups in R. When the <value expression> is applied to a given group of R, that group is the argument or argument source of each <set function specification> in the <value expression>. If SELECT_DISTINCT is not specified, then the multi-set is the result of the <subquery>. If SELECT_DISTINCT is specified, then the result of the <subquery> is the set of values derived from that multi-set by the elimination of any redundant duplicate values.

5.25. <query specification>

Function

Specify a table derived from the result of a <table expression>.

Format

```
<query specification> ::=
   [ SELEC | SELECT_ALL | SELECT_DISTINCT ]
      ( <select list> , <table expression> )

<select list> ::=
   <value expression> [ { & <value expression> } ... ]
 | '*'
```

Syntax Rules

1) The applicable <privilege>s for each <table name> contained in the <table expression> shall include SELEC. *Note: the "applicable <privileges>" for a <table name> will be defined in "<privilege definition>" for later levels of Ada/SQL. In Level 1, privileges are as defined to the underlying DBMS.*

2) Let R denote the result of the <table expression>.

3) The degree of the table specified by a <query specification> is equal to the cardinality of the <select list>.

4) The <select list> '*' is equivalent to a <value expression> sequence in which each <value expression> is a <column specification> that references a column of R and each column of R is referenced exactly once. The columns are referenced in the ascending sequence of their ordinal position within R.

5) Each <column specification> in each <value expression> shall unambiguously reference a column of R. A <query specification>, excluding any <subquery> contained within it, shall contain at most one use of an Ada/SQL keyword ending in DISTINCT (SELECT_DISTINCT, AVG_DISTINCT, MAX_DISTINCT, MIN_DISTINCT, SUM_DISTINCT, COUNT_DISTINCT).

6) If R is a grouped view, then the <select list> shall not contain a <set function specification>.

7) If R is a grouped table, then each <column specification> in each <value expression> shall reference a grouping column or be specified within a <set function specification>. If R is not a grouped table and any <value expression> includes a <set function specification>, then every <value expression> shall be specified within a <set function specification>.

8) Each column of the table that is the result of a <query specification> has the same data type as the <value expression> from which the column was derived.

9) If the n^{th} <value expression> in the <select list> consists of a single <column specification>, then the n^{th} column of the result is a named column whose <column name> is that of the <column specification>. Otherwise, the n^{th} column is an unnamed column.

10) A column of the table that is the result of a <query specification> is constrained to contain only nonnull values if and only if it is a named column that is constrained to contain only nonnull values.

11) A <query specification> is updatable if and only if the following conditions hold:

   a) SELECT_DISTINCT is not specified.

   b) Every <value expression> in the <select list> consists of a <column specification>.

   c) The <from clause> of the <table expression> specifies exactly one <table reference>, and that <table reference> refers to an updatable table.

   d) The <where clause> of the <table expression> does not include a <subquery>.

   e) The <table expression> does not include a <group by clause> or a <having clause>.

General Rules

1) If R is not a grouped table and the <select list> includes a <set function specification>, then R is the argument or argument source of each <set function specification> in the <select list> and the result of the <query specification> is a table consisting of one row. The nth value of the row is the value specified by the nth <value expression>.

2) If R is not a grouped table and the <select list> does not include a <set function specification>, then each <value expression> is applied to each row of R yielding a table of m rows, where m is the cardinality of R. The nth column of the table contains the values derived by the applications of the nth <value expression>. If SELECT_DISTINCT is not specified, then the table is the result of the <query specification>. If SELECT_DISTINCT is specified, then the result of the <query specification> is the table derived from that table by the elimination of any redundant duplicate rows.

3) If R is a grouped table that has zero groups and some <value expression> in the <select list> is a <column specification>, then the result of the <query specification> is an empty table. If R is a grouped table that has zero groups and every <value expression> in the <select list> is a <set function specification>, then the result of the <query specification> is a table having one row. The nth value in that row is the result of the nth <set function specification> in the <select list>.

4) If R is a grouped table that has one or more groups, then each <value expression> is applied to each group of R yielding a table of m rows, where m is the number of groups in R. The nth column of the table contains the values derived by the applications of the nth <value expression>. When a <value expression> is applied to a given group of R, that group is the argument or argument source of each <set function specification> in the <value expression>. If SELECT_DISTINCT is not specified, then the table is the result of the <query specification>. If SELECT_DISTINCT is specified, then the result of the <query specification> is the table derived from that table by the elimination of any redundant duplicate rows.

5) A row is a duplicate of another row if and only if all pairs of values with the same ordinal position are identical.

6. Schema Definition Language

6.1. <schema>

Function

Define a <schema>.

Format

```
<schema> ::= <schema compilation unit> ...

<schema compilation unit> ::=
        <authorization package compilation unit>
      | <schema package compilation unit>
```

Syntax Rules

1) The <schema compilation unit>s within a <schema> need not be part of the same compilation.

2) A <schema> shall contain exactly one <authorization package compilation unit>.

3) All <schema package compilation unit>s referencing the same <authorization package compilation unit> are part of the same <schema>.

General Rules

None.

6.1.1. <authorization package compilation unit>

Function

Define an <authorization identifier>.

Format

```
<authorization package compilation unit> ::=
    with SCHEMA_DEFINITION; use SCHEMA_DEFINITION;
    <authorization package specification> ;

<authorization package specification> ::=
    package  <identifier> is
     function <authorization identifier> is new AUTHORIZATION_IDENTIFIER;
    end [ <package simple name> ]

<package simple name> ::= <identifier>
```

Syntax Rules

1) The <package simple name>, if used within the <authorization package specification>, must repeat the <identifier>.

2) The <authorization identifier> shall be different from the <authorization identifier> of any other <schema> in the same environment.

3) An <authorization package specification> is said to define an authorization_package, and the <identifier> is taken as the name of that authorization package.

General Rules

None.

6.1.2. <schema package compilation unit>

Function

Define a portion of a <schema> that may be separately compiled.

Format

```
<schema package compilation unit> ::=
    <context clause>
    <schema package specification> ;

<schema package specification> ::=
    package <identifier> is
      [ <use clause> ... ]
      package ADA_SQL is
        [ <use clause> ... ]
        [ <schema authorization clause> ]
        [ <schema specification element> ... ]
      end ADA_SQL;
  end [ <package simple name> ]

<schema authorization clause> ::=
    SCHEMA_AUTHORIZATION : IDENTIFIER := <schema authorization identifier> ;

<schema authorization identifier> ::= <authorization identifier>

<schema specification element> ::=
    <type declaration>
  | <subtype declaration>
  | <table definition>
```

Syntax Rules

1) The <authorization identifier> shall match one directly contained within an <authorization package specification>. The <schema package compilation unit> is said to reference the <authorization package compilation unit> containing that <authorization package specification>.

2) A <with clause> and a <use clause> of the <context clause> shall each contain the name of the authorization package defining the <authorization identifier>.

3) If the schema package contains one or more table declarations, it shall then contain a <schema authorization clause>, and a <with clause> and a <use clause> of the <context clause> shall each name SCHEMA_DEFINITION,

4) The <package simple name>, if used within the <schema package specification>, must repeat the <identifier>.

5) The only <unit simple name>s that may be used within a <context clause> of a <schema package compilation unit> are those of schema packages, plus the following predefined packages: SCHEMA_DEFINITION, DATABASE.

Level 1 Implementation Rules.

1) *Each <schema package compilation unit> shall be placed in its own source file. The name of this source file shall be the same as that of the schema package defined, except that a system-dependent name augmentation may be used to indicate that the file contains Ada source code. This augmentation will be defined for each specific Ada/SQL system.*

General Rules

None.

6.1.3. <context clause>

Function

Specify other packages required by a package.

Format

```
<context clause> ::=
   [ { <with clause> [ <use clause> ... ] } ... ]

<with clause> ::=
   with <unit simple name> [ { , <unit simple name> } ... ] ;

<use clause> ::=
   use <package name> [ { , <package name> } ... ] ;
```

Syntax Rules

1) The form of <unit simple name> . ADA_SQL as a <package name> is permitted only if <unit simple name> is the name of a schema package, and is not permitted within a <context clause>.

2) A <unit simple name> contained within a <use clause> shall be contained within a textually prior <with clause>.

Level 1 Implementation Rules

1) *The file containing a schema package named in a <context clause> shall be accessible using only its name, without any other operating system dependent path information, at the time the <context clause> is processed by any Ada/SQL automated tool.*

General Rules

None.

6.1.4. <type declaration>

Function

Declare a data type.

Format

```
<type declaration> ::=
    <full type declaration>

<full type declaration> ::=
    type <type identifier> is <type definition> ;

<type definition> ::=
     <data type>
   | <derived type definition>
```

Syntax Rules

1) The <type identifier> shall not contain a _NOT_NULL or _NOT_NULL_UNIQUE suffix.

2) The <type identifier> is defined as the name of the type.

3) The <type identifier> shall be different from the <type identifier> or <table name> defined by any other <schema specification element> in the containing <schema package specification>.

General Rules

None.

6.1.5. <subtype declaration>

Function

Declare a data subtype.

Format

<subtype declaration> ::=
subtype <type identifier> **is** <subtype indication> ;

Syntax Rules

1) Case:

a) If the <type identifier> contains neither the _NOT_NULL nor the _NOT_NULL_UNIQUE suffix, then the <type identifier> within the <type mark> of the <subtype indication> shall also not contain either of these suffixes.

b) If the <type identifier> contains the _NOT_NULL suffix, then:

i) The <type identifier> within the <type mark> of the <subtype indication> shall not contain any suffix.

ii) The <identifier> of the <type identifier> within the <type mark> of the <subtype indication> shall match the <identifier> within the <type identifier>.

iii) The <subtype indication> shall not contain a <constraint>.

c) If the <type identifier> contains the _NOT_NULL_UNIQUE suffix, then:

i) The <type identifier> within the <type mark> of the <subtype indication> shall either contain no suffix or shall contain the _NOT_NULL suffix.

ii) The <identifier> of the <type identifier> within the <type mark> of the <subtype indication> shall match the <identifier> within the <type identifier>.

iii) The <subtype indication> shall not contain a <constraint>.

2) The <type identifier> is defined as the name of the subtype.

3) The <type identifier> shall be different from the <type identifier> or <table name> defined by any other <schema specification element> in the containing <schema package specification>.

General Rules

None.

6.2. <table definition>

Function

Define a table.

Format

```
<table definition> ::=
   type <table name> is
     record
       <table element> ...
     end record ;

<table element> ::= <column definition> ;
```

Syntax Rules

1) The <table name> shall not contain an <authorization identifier>.

2) The <table name> shall be different from the <table name> of any other <table definition> in the containing <schema>.

3) The <table name> shall be different from the <type identifier> or <table name> defined by any other <schema specification element> in the containing <schema package specification>.

4) The description of the table defined by a <table definition> includes the name <table name> and the column description specified by each <column definition>. The i^{th} column description is given by the i^{th} <column definition>.

Level 1 Implementation Rules

1) *A <table name> must match exactly the name of a table in the database.*

General Rules

1) A <table definition> defines either a base table or a viewed table. Which is defined depends on the definition of the table within the database.

6.3. <column definition>

Function

Define a column of a table.

Format

```
<column definition> ::=
    <column name> : <subtype indication>
```

Syntax Rules

1) The <column name> shall be different from the <column name> of any other <column definition> in the containing <table definition>.

2) The n^{th} column of the table is described by the n^{th} <column definition> in the <table definition>.

3) The description of the column defined by a <column definition> includes the name <column name> and the data type specified by the <type mark> and optional <constraint> of the <subtype indication>.

Level 1 Implementation Rules

1) *The <column name> shall match the name of the corresponding column in the database table whose name is given by the <table name> of the enclosing <table definition>.*

2) *Case:*

   a) *If the <type identifier> of the <type mark> of the <subtype indication> contains neither the _NOT_NULL nor the _NOT_NULL_UNIQUE suffix, then the corresponding database column shall not have a NOT NULL or a NOT NULL UNIQUE constraint placed on it.*

   b) *If the <type identifier> of the <type mark> of the <subtype indication> contains the _NOT_NULL suffix, then the corresponding database column shall be constrained to contain only NOT NULL values.*

   c) *If the <type identifier> of the <type mark> contains the _NOT_NULL_UNIQUE suffix, then the corresponding database column shall be constrained to contain only NOT NULL UNIQUE values.*

General Rules

1) If a column is constrained to contain only nonnull or unique values, then the constraint is effectively checked after the execution of each <Ada/SQL statement>.

6.4. *<unique constraint definition>*

6.5. *<view definition>*

6.6. *<privilege definition>*

These items are not defined in Level 1 Ada/SQL. They must be defined for the database using DBMS facilities before Ada/SQL programs are run against the database.

7. Program Environment

7.1. <program environment>

Function

Define the program environment of an Ada/SQL program.

Format

```
<program environment> ::=
    <Ada/SQL package compilation unit> ...
    <authorization package compilation unit> ...
    <schema package compilation unit> ...
    <Ada/SQL variable package compilation unit> ...

<Ada/SQL package compilation unit> ::=
    <Ada/SQL context clause>
    [ <Ada context clause> ]
    <library unit body>

<Ada/SQL context clause> ::=
    <with clause>
    [ <use clause> ]
```

Syntax Rules

1) The <with clause> and optional <use clause> of the <Ada/SQL context clause> shall name all <schema package compilation unit>s and <Ada/SQL variable package compilation unit>s necessary for the *<Ada/SQL compilation unit>*.

2) The <library unit body> shall consist of regular Ada statements and <Ada/SQL DML statements>.

General Rules

None.

7.2. <Ada/SQL variable package compilation unit>

Function

Define the program variables that will be used in Ada/SQL DML statements.

Format

<Ada/SQL variable package compilation unit> ::=
  <context clause>
  <Ada/SQL variable package specification> ;

<Ada/SQL variable package specification> ::=
  package <identifier> **is**
   [<use clause> ...]
   [<variable declarations> ...]
  end [<package simple name>]

Syntax Rules

1) The only <unit simple name>s that may be used within a <context clause> of a <Ada/SQL variable package compilation unit> are those of schema packages, plus the predefined package CURSOR_DEFINITION.

2) The <variable declarations> shall declare program variables to be used with the Ada/SQL DML statements. All <variable declarations> must be of types defined in <schema package compilation units> or of type CURSOR_NAME defined in package CURSOR_DEFINITION.

3) The <package simple name>, if used within the <Ada/SQL variable package specification>, must repeat the <identifier>.

Level 1 Implementation Rules.

1) *Each <Ada/SQL variable package compilation unit> shall be placed in its own source file. The name of this source file shall be the same as that of the variable package defined, except that a system-dependent name augmentation may be used to indicate that the file contains Ada source code. This augmentation will be defined for each specific Ada/SQL system.*

General Rules

None.

8. Data Manipulation Language

8.1. <close statement>

Function

Close a cursor.

Format

```
<close statement> ::=
    CLOSE ( <cursor name> ) ;
```

Syntax Rules

1) The <cursor name> is a program variable of type CURSOR_NAME, which is a private type defined by the implementation.

General Rules

1) The cursor, CR, named by <cursor name>, shall be in the open state.

2) Cursor CR is placed in the closed state and the copy of the <cursor specification> of CR is destroyed.

8.2. <commit statement>

Function

Terminate the current transaction with commit.

Format

```
<commit statement> ::=
    COMMIT_WORK ;
```

Syntax Rules

None.

Level 1 Implementation Rules

1) *<commit statement>s shall not be used with UNIFY -- transactions are not supported.*

General Rules

1) The current transaction is terminated.

2) Any cursors that were opened by the current transaction are closed.

3) Any changes to the database that were made by the current transaction are committed.

8.3. <declare cursor>

Function

Define a cursor.

Format

```
<declare cursor> ::=
   DECLAR ( <cursor name> , CURSOR_FOR => <cursor specification> ) ;

<cursor specification> ::=
   <query expression> [ , <order by clause> ]

<query expression> ::=
   <query term>
 | <query expression> & { UNION | UNION_ALL } ( <query term> )
 | <query expression> & { UNION | UNION_ALL }   <query term>

<query term> ::=
   <query specification>
 | ( <query expression> )

<order by clause> ::=
   ORDER_BY => <sort specification> [ { & <sort specification> } ... ]

<sort specification> ::=
       <sort column specification>
 | ACS  ( <sort column specification> )
 | DESC ( <sort column specification> )

<sort column specification> ::=
   <column number>
 | <column specification>
```

Syntax Rules

1) The <cursor name> is the cursor to be declared, a program variable of type CURSOR_NAME, which is a private type defined by the implementation.

2) If a <query expression> including a UNION or UNION_ALL contains an unparenthesized <query term>, then that <query term> shall be of the form (<query expression>).

3) A <column number> shall be a positive integer of type COLUMN_NUMBER.

4) Let T denote the table specified by the <cursor specification>.

5) Case:

   a) If ORDER_BY is specified, then T is a read-only table with the specified sort order.

   b) If neither ORDER_BY, UNION, nor UNION_ALL is specified and the <query specification> is updatable, then T is an updatable table.

   c) Otherwise, T is a read-only table.

6) Case:

   a) If neither UNION nor UNION_ALL is specified, then the description of T is the description of the <query specification>.

   b) If UNION or UNION_ALL is specified, then for each UNION or UNION_ALL that is specified, except for <column name>s the description of the table specified by the first <query expression>

and the <query term> shall be identical. All columns of the result are unnamed. Except for <column name>s the description of the result is the same as the description of the tables specified by the first <query expression> and the <query term>.

7) If ORDER_BY is specified, then each <column specification> in the <order by clause> shall identify a column of T, and each <column number> in the <order by clause> shall be greater than zero and not greater than the degree of T. A named column may be referenced by a <column number> or a <column specification>. An unnamed column shall be referenced by a <column number>.

Level 1 Implementation Rules

1) *No UNIONS shall be used with UNIFY.*

2) *No <column numbers> shall be used as <sort column specification>s with UNIFY.*

3) *No <order by clause> shall include GROUP_BY in UNIFY.*

General Rules

1) The <cursor name> shall not name an open cursor.

2) Case:

a) If T is an updatable table, then the cursor is associated with the named table identified by the <table name> in the <from clause>. Let B denote that named table. For each row in T, there is a corresponding row in B from which the row of T is derived. When the cursor is positioned on a row of T, the cursor is also positioned on the corresponding row of B.

b) Otherwise, the cursor is not associated with a named table.

3) Whether T is a temporary viewed table or a temporary base table is is defined by the underlying DBMS. Any program relying on this is erroneous.

4) Case:

a) If T is a temporary viewed table, then a row of T is derived only when the cursor is positioned on that row by a <fetch statement>.

b) If T is a temporary base table, then T is a temporary table that is created when the cursor is opened. A temporary table persists until the cursor is closed.

5) Case:

a) If neither UNION nor UNION_ALL is specified, then T is the result of the specified <query specification>.

b) If UNION or UNION_ALL is specified, then for each UNION or UNION_ALL that is specified let T1 and T2 be the result of the <query expression> and the <query term>. The result of the UNION or UNION_ALL is derived as follows:

i) Initialize the result to an empty table.

ii) Insert each row of T1 and each row of T2 into the result.

iii) If UNION_ALL is not specified, then eliminate any redundant duplicate rows from the result.

6) Case:

a) If ORDER_BY is not specified, then the ordering of rows in T is defined by the underlying DBMS. This order is subject to the reproducibility requirement within a transaction, but it may change between transactions. Any program relying on this order is erroneous.

b) If ORDER_BY is specified, then T has a sort order:

i) The sort order is a sequence of sort groups. A sort group is a sequence of rows in which all values of a sort column are identical. Furthermore, a sort group may be a sequence of sort groups.

ii) The cardinality of the sequence and the ordinal position of each sort group is determined by the values of the most significant sort column. The cardinality of the sequence is the minimum number of sort groups such that, for each sort group of more than one row, all values of that sort column are identical.

iii) If the sort order is based on additional sort columns, then each sort group of more than one row is a sequence of sort groups. The cardinality of each sequence and the ordinal position of each sort group within each sequence is determined by the values of the next most significant sort column. The cardinality of each sequence is the minimum number of sort groups such that, for each sort group of more than one row, all values of that sort column are identical.

iv) The preceding paragraph applies in turn to each additional sort column. If a sort group consists of multiple rows and is not a sequence of sort groups, then the order of the rows within that sort group is assigned by the underlying DBMS. Any program relying on this order is erroneous.

v) Let C be a sort column and let S denote a sequence which is determined by the values of C.

vi) A sort direction is associated with each sort column. If the direction of C is ascending, then the first sort group of S contains the lowest value of C and each successive sort group contains a value of C that is greater than the value of C in its predecessor sort group. If the direction is descending, then the first sort group of S contains the highest value of C and each successive sort group contains a value of C that is less than the value of C in its predecessor sort group.

vii) Ordering is determined by the comparison rules specified in 5.11 "<comparison predicate>". The order of the null value relative to nonnull values is defined in the underlying DBMS, but shall be either greater than or less than all nonnull values. Any program relying on this order is erroneous.

viii) A <sort specification> specifies a sort column and a direction. The sort column is the column referenced by the <column number> or the <column specification>. The <column number> n references the n^{th} column of R. A <column specification> references the named column.

ix) If DESC is specified in a <sort specification>, then the direction of the sort column specified by that <sort specification> is descending. If ASC is specified or if neither ASC or DESC is specified, then the direction of the sort column is ascending.

x) The <sort specification> sequence determines the relative significance of the sort columns. The sort column specified by the first <sort specification> is the most significant sort column and each successively specified sort column is less significant than the previously specified sort column.

8.4. <delete statement: positioned>

Function

Delete a row of a table based on the current position of a cursor.

Format

```
<delete statement: positioned> ::=
    DELETE_FROM ( <table name> , WHERE_CURRENT_OF => <cursor name> ) ;
```

Syntax Rules

1) The applicable <privilege>s for each <table name> contained in the <table expression> shall include DELETE. *Note: the "applicable <privileges>" for a <table name> will be defined in "<privilege definition>" for later levels of Ada/SQL. In Level 1, privileges are as defined to the underlying DBMS.*

2) The table designated by the named cursor, CR, shall not be a read-only table.

3) Let T denote the table identified by the <table name>. T shall be the table identified in the first <from clause> in the <cursor specification> of CR.

General Rules

1) The containing program environment shall contain a <declare cursor> CR whose <cursor name> is the same as the <cursor name> in the <delete statement: positioned>.

2) Cursor CR shall be open and positioned on a row.

3) The row from which the current row of CR is derived is deleted.

8.5. <delete statement: searched>

Function

Delete rows of a table based on a search criterion.

Format

```
<delete statement: searched> ::=
   DELETE_FROM ( <table name> [ , WHERE => <search condition> ] ) ;
```

Syntax Rules

1) The applicable <privilege>s for each <table name> contained in the <table expression> shall include DELETE. *Note: the "applicable <privileges>" for a <table name> will be defined in "<privilege definition>" for later levels of Ada/SQL. In Level 1, privileges are as defined to the underlying DBMS.*

2) Let T denote the table identified by the <table name>. T shall not be a read-only table or a table that is identified in a <from clause> of any <subquery> contained in the <search condition>.

3) The scope of the <table name> is the entire <delete statement: searched>.

General Rules

1) Case:

   a) If a <search condition> is not specified, then all rows of T are deleted.

   b) If a <search condition> is specified, then it is applied to each row of T with the <table name> bound to that row, and all rows for which the result of the <search condition> is true are deleted. Each <subquery> in the <search condition> is effectively executed for each row of T and the results used in the application of the <search condition> to the given row of T. If any executed <subquery> contains an outer reference to a column of T, the reference is to the value of that column in the given row of T. Note: "outer reference" is defined in 5.7 "<column specification>".

8.6. <fetch statement>

Function

Position a cursor on the next row of a table and assign values in that row to program variables.

Format

```
<fetch statement> ::=
   FETCH ( <cursor name> );
   INTO ( <result specification> [ , <cursor name> ] ) ;
 [ { INTO ( <result specification> [ , <cursor name> ] ) ; } ... ]

<result specification> ::=
   <result program variable>
   [ , <last variable> ] [ , <indicator variable> ]
```

Syntax Rules

1) Let CR be the <declare cursor> whose <cursor name> is the same as the <cursor name> of the <fetch statement>. Let T be the table defined by the <cursor specification> of CR.

2) A FETCH procedure call shall be followed by as many calls to INTO as are required to retrieve the values of each column in the row. Each INTO returns one column value.

3) The <result program variable> of the i^{th} INTO statement shall be a program variable to obtain column values from the i^{th} column of T. The i^{th} <result program variable>s shall be of the same type as that of the i^{th} column of T being retrieved.

4) The <last variable> shall be a program variable to obtain the value of the last index position used in retrieving array values (strings). It is used when and only when <result program variable> is of type array. For one dimensional arrays, which all strings are, <last variable> is of the same type as the array index.

5) The <indicator variable> shall be an optional program variable of type INDICATOR_VARIABLE, set to NULL_VALUE if the database column retrieved contains a null_value, else set to NOT_NULL. If a null value is retrieved from the database but no <indicator variable> is specified, the NULL_ERROR exception will be raised.

6) Case:

   a) If several tasks within the same program are simultaneously performing database retrievals, the <cursor name> used in the FETCH must be specified as the final parameter to INTO procedures for that FETCH.

   b) If simultaneous database retrievals are not being performed, the <cursor name> parameter may be omitted from the INTO calls.

General Rules

1) The containing program environment shall contain a <declare cursor> CR whose <cursor name> is the same as the <cursor name> of the <fetch statement>. This cursor shall be in the open state.

2) If the table designated by cursor CR is empty or if the position of CR is on or after the last row, CR is positioned after the last row, the exception NOT_FOUND_ERROR will be raised and values are not assigned to the program variables identified by the <result specification>.

3) The NOT_FOUND_ERROR exception will be raised if a FETCH is performed on a cursor for which all rows (if any) have already been returned.

4) If the position of CR is before a row, CR is positioned on that row and values in that row are assigned to their corresponding program variables.

5) If the position of CR is on r, where r is a row other than the last row, CR is positioned on the row

immediately after r and values in the row immediately after r are assigned to their corresponding program variables.

6) The order of the assignment of values to program variables is the same as the ordering of the INTO calls.

7) If an error occurs during the assignment of a value to a program variable, the values returned by the INTO statements are undefined. Any program relying on these values is erroneous.

8) The exception CONSTRAINT_ERROR will be raised if the result will not fit in the subtype of the <result program variable>.

9) If the result is a string whose length is less than the length of the string of the <result program variable>, <last variable> will be set accordingly and the index positions of <result program variable> beyond <last variable> will not be altered.

8.7. <insert statement>

Function

Create new rows in a table.

Format

```
<insert statement> ::=
   INSERT_INTO ( <table name> [ ( <insert column list> ) ] ,
  { VALUES <= <insert value list> } | <query specification> ) ;

<insert column list> ::=
   <column name> [ { & <column name> } ... ]

<insert value list> ::=
   <insert value> [ { AND <insert value> } ... ]

<insert value> ::=
   <value specification>
  | NULL_VALUE
```

Syntax Rules

1) The applicable <privilege>s for each <table name> contained in the <table expression> shall include INSERT. *Note: the "applicable <privileges>" for a <table name> will be defined in "<privilege definition>" for later levels of Ada/SQL. In Level 1, privileges are as defined to the underlying DBMS.*

2) Let T denote the table identified by the <table name>. T shall not be a read-only table or a table that is identified in a <from clause> of the <query specification> or of any <subquery> contained in the <query specification>.

3) If an <insert column list> is used and the <table name> includes an <authorization identifier>, then the syntax for the <table name> is <authorization identifier>-<table identifier>. This is one of the three contexts within Ada/SQL where <table name> syntax is not the usual <authorization identifier>.<table name>.

4) Each <column name> in the <insert column list> shall identify a column of T and the same column shall not be identified more than once. Omission of the <insert column list> is an implicit specification of a <insert column list> that identifies all columns of T in the ascending sequence of their ordinal position within T.

5) A column identified by the <insert column list> is an object column.

6) Case:

   a) If an <insert value list> is specified, then the number of <insert value>s in that <insert value list> shall be equal to the number of <column name>s in the <insert column list>. Let the i^{th} item of the <insert statement> refer to the i^{th} <value specification> in that <insert value list>.

   b) If a <query specification> is specified, then the degree of the table specified by that <query specification> shall be equal to the number of <column name>s in the <insert column list>. Let the i^{th} item of the <insert statement> refer to the i^{th} column of the table specified by the <query specification>.

7) If the i^{th} item of the <insert statement> is not the <insert value> NULL_VALUE, then, the data type of the column of table T designated by the i^{th} <column name> shall be the same as the data type of the i^{th} item of the <insert statement>.

Level 1 Implementation Rules

1) *NULL VALUE may not be used with UNIFY.*

2) *With UNIFY, a column which would otherwise be set to a null value is instead set to a default initial*

value, based on its type.

3) *Constraint checking will not be performed by Level 1.*

General Rules

1) A row is inserted in the following steps:

   a) A candidate row is effectively created in which the value of each column is the null value. If T is a base table, B, then the candidate row includes a null value for every column of B. If T is a viewed table, the candidate row includes a null value for every column of the base table, B from which T is derived.

   b) For each object column in the candidate row, the value is replaced by an insert value.

   c) The candidate row is inserted in B.

2) If T is a viewed table defined by a <view definition> that specifies "WITH CHECK OPTION", then if the <query specification> contained in the <view definition> specified a <where clause> that is not contained in a <subquery>, then the <search condition> of that <where clause> shall be true for the candidate row.

3) If an <insert value list> is specified, then case:

   a) If the ith <insert value> of the <insert value list> is a <value specification>, then the ith value of the candidate row is the value of that <value specification>.

   b) If the ith <insert value> of the <insert value list> is NULL_VALUE, then the ith value of the candidate row is the null value.

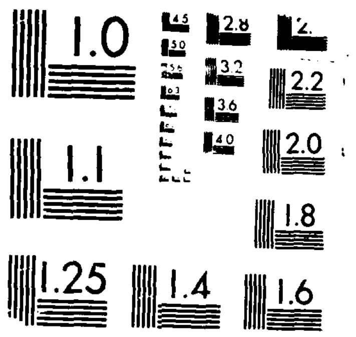
4) If a <query specification> is specified, then let R be the result of the <query specification>. If R is empty, then the exception NOT_FOUND_ERROR is raised and no row is inserted. The number of candidate rows created is equal to the cardinality of R. The insert values of one candidate row are the values in one row of R and the values in one row of R are the insert values of one candidate row.

5) Let V denote a row of R or the sequence of values specified by the <value list>. The nth value of V is the insert value of the object column identified by the nth <column name> in the <insert column list>.

6) If an <insert value> is a string which is longer than the <column name> can hold, then <insert value> will be truncated.

7) If an <insert value> is a string which is shorter than the <column name> can hold, then <insert value> will be padded with spaces.

AD-A194 400   LEVEL 1 ADA/SQL (STRUCTURED QUERY LANGUAGE) DATABASE   2/2
LANGUAGE INTERFACE U..(U) INSTITUTE FOR DEFENSE
ANALYSES ALEXANDRIA VA B BRYKCZYNSKI ET AL. SEP 87
UNCLASSIFIED   IDA-M-360 IDA/HQ-87-32693 MDA903-84-C-0031 F/G 12/5   NL



MICROCOPY RESOLUTION TEST CHART

REAU TANDARDS 1963-A

8.8. <open statement>

Function

Open a cursor.

Format

```
 <open statement> ::=
    OPEN ( <cursor name> ) ;
```

Syntax Rules

None.

General Rules

1) A previously executed <declare cursor> shall have associated a <cursor specification> with <cursor name> CR.

2) Cursor CR shall be in the closed state.

3) Let S denote the <cursor specification> of cursor CR.

4) Cursor CR is opened in the following steps:

   a) If S specifies a read_only table, then that table, as specified by the copy of S, is effectively created.

   b) Cursor CR is placed in the open state and its position is before the first row of the table.

8.9. <rollback statement>

Function

Terminate the current transaction with rollback.

Format

```
<rollback statement> ::=
   ROLLBACK_WORK ;
```

Syntax Rules

None.

Level 1 Implementation Rules

1) *<rollback statement>s shall not be used with UNIFY.*

General Rules

1) Any changes to the database that were made by the current transaction are canceled.

2) Any cursors that were opened by the current transaction are closed.

3) The current transaction is terminated.

8.10. <select statement>

Function

Specify a table and assign the values in the single row of that table to program variables.

Format

```
 <select statement> ::=
    [ SELEC | SELECT_ALL | SELECT_DISTINCT ] ( <select list> ,
      <table expression> ) ;
      INTO ( <result specification> ) ;
   [ { INTO ( <result specification> ) ; } ... ]
```

Syntax Rules

1) The applicable <privilege>s for each <table name> contained in the <table expression> shall include SELEC. *Note: the "applicable <privileges>" for a <table name> will be defined in "<privilege definition>" for later levels of Ada/SQL. In Level 1, privileges are as defined to the underlying DBMS.*

2) The <table expression> shall not include a <group by clause> or a <having clause> and shall not identify a grouped view.

3) The number of elements in the <select list> shall be the same as the number of following INTO calls.

4) The data type of the <result program variable> of the <result specification> in the i^{th} INTO call shall be the same as the data type of the i^{th} <value expression> in the <select list>.

5) The <last variable> within <result specification> shall be a program variable to obtain the value of the last index position used in retrieving array values (strings). It is used when and only when <result program variable> within <result specification> is of type array. For one dimensional arrays, which all strings are, <last variable> within <result specification> is of the same type as the array index.

6) The <indicator variable> within the <result specification> shall be an optional program variable of type INDICATOR_VARIABLE, set to NULL_VALUE if the database column retrieved contains a null_value, else set to NOT_NULL. If a null value is retrieved from the database but no <indicator variable> within <result specification> is specified, the NULL_ERROR exception will be raised.

General Rules

1) A <cursor name> is not defined in the <select statement>. Consequently, tasks within a program must not perform more than one simultaneous <select statement>. If multiple retrievals must be performed simultaneously, FETCH statements must be used to avoid erroneous results.

2) Let S be a <query specification> whose <select list> and <table expression> are those specified in the <select statement> and which specifies SELECT_ALL or SELECT_DISTINCT if it is specified in the <select statement>. Let R denote the result of <query specification> S.

3) The cardinality of R shall not be greater than one. The UNIQUE_ERROR exception will be raised if the <select statement> retrieved more than one row, resulting in the cardinality of R being greater than one. If R is empty, then the NOT_FOUND_ERROR exception will be raised.

4) If R is not empty, then values in the row of R are assigned to their corresponding program variables.

5) The order of the assignment of values to program variables is the same as the ordering of the INTO calls.

6) If an error occurs during the assignment of a value to a program variable, the values returned by the INTO statements are undefined. Any program relying on these values is erroneous.

7) The <result specification> of the n^{th} INTO calls, corresponds to the n^{th} value in the row of R.

8) The exception CONSTRAINT_ERROR will be raised if the result will not fit in the subtype of the <result program variable>.

9) If the result is a string whose length is less than the length of the string of the <result program variable>, <last variable> will be set accordingly and the index positions of <result program variable> beyond <last variable> will not be altered.

8.11. <update statement: positioned>

Function

Modify a row of a table based on a cursor's current position.

Format

```
<update statement: positioned> ::=
   UPDATE ( <table name> ,
   SET => <set clause> [ { AND <set clause> } ... ] ,
   WHERE_CURRENT_OF => <cursor name> ) ;

<set clause> ::=
   <object column> <= { <value expression> | NULL_VALUE }

<object column> ::=
   <column name>
```

Syntax Rules

1) The applicable <privilege>s for each <table name> contained in the <table expression> shall include UPDATE. *Note: the "applicable <privileges>" for a <table name> will be defined in "<privilege definition>" for later levels of Ada/SQL. In Level 1, privileges are as defined to the underlying DBMS.*

2) The table designated by CR shall not be a read-only table.

3) Let T denote the table identified by the <table name>. T shall be the table identified in the first <from clause> in the <cursor specification> of CR.

4) A <value expression> in a <set clause> shall not include a <set function specification>.

5) The same <object column> shall not appear more than once in an <update statement: positioned>.

6) For each <set clause>, case:

   a) If NULL_VALUE is specified, then the column designated by the <object column> shall allow nulls.

   b) The data type of the column designated by the <object column> shall be the same as the data type of the <value expression> for that column.

Level 1 Implementation Rules

1) *<update statement: positioned> shall not be used with UNIFY.*

2) *Subtype constraint checking will not be performed by Level 1.*

General Rules

1) Cursor CR shall be opened and positioned on a row.

2) The object row is that row from which the current row of CR is derived.

3) The object row is updated as specified by each <set clause>. A <set clause> specifies an object column and an update value of that column. The object column is the column identified by the <object column> in the <set clause>. The update value is the null value or the value specified by the <value expression>. If the <value expression> contains a reference to a column of T, the reference is to the value of that column in the object row before any value of the object row is updated.

4) The object row is updated in the following steps:

   a) A candidate row is created which is a copy of the object row.

   b) For each <set clause>, the value of the specified object column in the candidate row is replaced by the specified update value.

c) The object row is replaced by the candidate row.

5) If T is a viewed table defined by a <view definition> that specifies "WITH CHECK OPTION", then if the <query specification> contained in the <view definition> specifies a <where clause> that is not contained in a <subquery>, then the <search condition> of that <where clause> shall be true for the candidate row.

6) The containing program environment shall contain a <declare cursor> CR where <cursor name> is the same as the <cursor name> in the <update statement: positioned>.

7) If a <set clause> is a string which is longer than the column of the <table name> being updated, then the <set clause> will be truncated.

8) If a <set clause> is a string which is shorter than the column of the <table name> being updated, then the <set clause> will be padded with spaces.

8.12. <update statement: searched>

Function

Modify rows of a table based on a search condition.

Format

```
<update statement: searched> ::=
   UPDATE ( <table name> ,
   SET => <set clause> [ { AND <set clause> } ... ] ,
   WHERE => <search condition> ) ;
```

Syntax Rules

1) The applicable <privilege>s for each <table name> contained in the <table expression> shall include UPDATE. *Note: the "applicable <privileges>" for a <table name> will be defined in "<privilege definition>" for later levels of Ada/SQL. In Level 1, privileges are as defined to the underlying DBMS.*

2) Let T denote the table identified by the <table name>. T shall not be a read-only table or a table that is identified in a <from clause> of any <subquery> that is contained in the <search condition>.

3) A <value expression> in a <set clause> shall not include a <set function specification>.

4) The same <object column> shall not appear more than once in an <update statement: searched>.

5) The scope of the <table name> is the entire <update statement: searched>.

6) For each <set clause>, case:

   a) If NULL_VALUE is specified, then the column designated by the <object column> shall allow nulls.

   b) The data type of the column designated by the <object column> shall be the same as the data type of the <value expression> for that column.

Level 1 Implementation Rules

1) *NULL_VALUES shall not be used in a <set clause> with UNIFY.*

2) *Subtype constraint checking will not be performed by Level 1.*

General Rules

1) Case:

   a) If a <search condition> is not specified, then all rows of T are the object rows.

   b) If a <search condition> is specified, then it is applied to each row of T with the <table name> bound to that row, and the object rows are those rows for which the result of the <search condition> is true. Each <subquery> in the <search condition> is effectively executed for each row of T and the results used in the application of the <search condition> to the given row of T. If any executed <subquery> contains an outer reference to a column of T, the reference is to the value of that column in the given row of T. Note: "outer reference" is defined in 5.7 "<column specification>".

2) Each object row is updated as specified by each <set clause>. A <set clause> specifies an object column and a update value of that column. The object column is the column identified by the <object column> in the <set clause>. The update value is the null value or the value specified by the <value expression>. If the <value expression> contains a reference to a column of T, the reference is to the value of that column in the object row before any value of the object row is updated.

3) The object row is updated in the following steps:

   a) A candidate row is created which is a copy of the object row.

b) For each <set clause>, the value of the specified object column in the candidate row is replaced by the specified update value.

c) The object row is replaced by the candidate row.

4) If T is a viewed table defined by a <view definition> that specifies "WITH CHECK OPTION", then if the <query specification> contained in the <view definition> specifies a <where clause> that is not contained in a <subquery>, then the <search condition> of that <where clause> shall be true for the candidate row.

5) If a <set clause> is a string which is longer than the column of the <table name> being updated, then the <set clause> will be truncated.

6) If a <set clause> is a string which is shorter than the column of the <table name> being updated, then the <set clause> will be padded with spaces.

Distribution List for IDA Memorandum Report M-360

NAME AND ADDRESS	NUMBER OF COPIES
Sponsor	
Ms. Sally Barnes HQ Defense Logistics Agency (DLA) ATTN DLA-ZWS Cameron Station Alexandria, VA 22304-6100	10 copies
Other	
Defense Technical Information Center Cameron Station Alexandria, VA 22314	2 copies
IIT Research Institute 4550 Forbes Blvd., Suite 300 Lanham, MD 20706	1 copy
Mr. Fred Friedman P.O. Box 576 Annandale, VA 22003	1 copy
Ms. Patty Hicks DSAC-SR 3990 Broad St. Columbus, OH 43216-5002	1 copy
Ms. Kerry Hilliard 7321 Franklin Road Annandale, VA 22003	1 copy
Ms. Elinor Koffee DSAC-SR 3990 Broad St. Columbus, OH 43216-5002	1 copy
CSED Review Panel	
Dr. Dan Alpert, Director Center for Advanced Study University of Illinois 912 W. Illinois Street Urbana, Illinois 61801	1 copy

NAME AND ADDRESS	NUMBER OF COPIES
Dr. Barry W. Boehm TRW Defense Systems Group MS 2-2304 One Space Park Redondo Beach, CA 90278	1 copy
Dr. Ruth Davis The Pymatuning Group, Inc. 2000 N. 15th Street, Suite 707 Arlington, VA 22201	1 copy
Dr. Larry E. Druffel Software Engineering Institute Shadyside Place 480 South Aiken Av. Pittsburgh, PA 15231	1 copy
Dr. C.E. Hutchinson, Dean Thayer School of Engineering Dartmouth College Hanover, NH 03755	1 copy
Mr. A.J. Jordano Manager, Systems & Software Engineering Headquarters Federal Systems Division 6600 Rockledge Dr. Bethesda, MD 20817	1 copy
Mr. Robert K. Lehto Mainstay 302 Mill St. Occoquan, VA 22125	1 copy
Mr. Oliver Selfridge 45 Percy Road Lexington, MA 02173	1 copy

IDA

General W.Y. Smith, HQ	1 copy
Mr. Seymour Deitchman, HQ	1 copy
Mr. Philip Major, HQ	1 copy
Dr. Jack Kramer, CSED	1 copy
Dr. Robert I. Winner, CSED	1 copy
Dr. John Salasin, CSED	1 copy
Mr. Bill Brykczynski, CSED	10 copies
Ms. Audrey A. Hook, CSED	2 copies
Ms. Katydean Price, CSED	2 copies
IDA Control & Distribution Vault	3 copies

END
DATE
FILMED
8-88
DTIC