UNISYS

Product Information Announcement

o New Release ● Revision o Update o New Mail Code

Title

MCP/AS InfoExec™ Semantic Information Manager (SIM) Programming Guide (8600 1666–002)

This announces a retitling and reissue of the *ClearPath HMP NX and A Series InfoExec™ Semantic Information Manager (SIM) Programming Guide*. No new technical changes have been introduced since the HMP 1.0 and SSR 43.2 release in June 1996.

To order a Product Information Library CD-ROM or paper copies of this document

- United States customers, call Unisys Direct at 1-800-448-1424.
- Customers outside the United States, contact your Unisys sales office.
- Unisys personnel, order through the electronic Book Store at http://iwww.bookstore.unisys.com.

Comments about documentation can be sent through e-mail to **doc@unisys.com**.

Announcement only:

Announcement and attachments:
AS220

System: MCP/AS
Release: HMP 4.0 and SSR 45.1
Date: June 1998
Part number: 8600 1666–002

MCP/AS

UNISYS

InfoExec™ Semantic Information Manager (SIM)

Programming Guide

HMP 4.0 and SSR 45.1 June 1998

Printed in USA
Priced Item 8600 1666–002

The names, places, and/or events used in this publication are not intended to correspond to any individual, group, or association existing, living, or otherwise. Any similarity or likeness of the names, places, and/or events with the names of any individual, living or otherwise, or that of any group or association is purely coincidental and unintentional.

NO WARRANTIES OF ANY NATURE ARE EXTENDED BY THIS DOCUMENT. Any product or related information described herein is only furnished pursuant and subject to the terms and conditions of a duly executed agreement to purchase or lease equipment or to license software. The only warranties made by Unisys, if any, with respect to the products described in this document are set forth in such agreement. Unisys cannot accept any financial or other responsibility that may be the result of your use of the information in this document or software material, including direct, special, or consequential damages.

You should be very careful to ensure that the use of this information and/or software material complies with the laws, rules, and regulations of the jurisdictions with respect to which it is used.

The information contained herein is subject to change without notice. Revisions may be issued to advise of such changes and/or additions.

RESTRICTED – Use, reproduction, or disclosure is restricted by DFARS 252.227–7013 and 252.211–7015/FAR 52.227–14 & 52.227-19 for commercial computer software.

Correspondence regarding this publication should be forwarded to Unisys Corporation either by using the Business Reply Mail form at the back of this document or by addressing remarks to Software Product Information, Unisys Corporation, 25725 Jeronimo Road, Mission Viejo, CA 92691–2792 U.S.A.

Comments about documentation can also be sent through e-mail to **doc@unisys.com**.

Contents

Tables

About This Guide

Purpose

This document describes how to query Semantic Information Manager (SIM) databases through COBOL74, ALGOL, or Pascal programs.

Scope

This guide describes only concepts about using programming to access SIM databases. For specific statement syntax and explanations, refer to the *ALGOL Programming Reference Manual*, *Volume 2: Product Interfaces*, the *COBOL ANSI-74 Programming Reference Manual*, *Volume 2: Product Interfaces*, or the *Pascal Programming Reference Manual*, *Volume 2: Product Interfaces*.

Audience

This guide is written for programmers experienced with COBOL74, ALGOL, or Pascal, and with InfoExec SIM concepts

Prerequisites

This guide assumes that you are familiar with programming and InfoExec SIM concepts and that you have read the *InfoExec Capabilities Manual* and the *InfoExec Semantic Information Manager (SIM) Technical Overview*. Additionally, you can refer to the *InfoExec Semantic Information Manager (SIM) Object Manipulation Language (OML) Programming Guide* for more explanation of SIM concepts.

How to Use This Guide

This guide describes query program statements generically. If all the letters of a statement are capitalized in a paragraph, the statement is the same in COBOL74, ALGOL, and Pascal. If a statement is in lowercase letters, the statement differs in each language. A table in Appendix B, “Language Comparison Tables,” illustrates how such statements appear in COBOL74, ALGOL, and Pascal. For exact statement syntax, consult the appropriate language manual.

Examples in this guide use the ORGANIZATION database, which is described in the *InfoExec SIM Technical Overview*. Although this guide describes program queries generically, it uses COBOL74 program fragments as examples. Note that COBOL74

automatically qualifies attributes and their classes, or classes and their databases. COBOL74, ALGOL, and Pascal program examples can be found in Appendix A, “Sample Programs.”

Organization

This guide consists of the following sections and appendixes. In addition, an index appears at the end of this guide.

Section 1. Programming in the InfoExec Environment

This section describes how to query SIM databases through programs. It then discusses other program extensions that can access the data dictionary.

Section 2. Starting a Query Program

Before writing a query statement, you must declare the query and the database used in the statement. This section shows how to declare databases and queries and how to open the database. It also describes how to write a query statement.

Section 3. Choosing Entities

Through selection expressions, you can choose specific entities and their values to process. This section describes the expressions that select one or more entities.

Section 4. Retrieving Entities

You retrieve entities through retrieval queries that consist of SELECT and RETRIEVE statements. This section describes how to use these statements to assign entity values to attributes and how to format entities.

Section 5. Designating Transaction State

A program can update SIM databases only when it is in transaction state. This section describes how to control transaction state in a program.

Section 6. Updating Entities

You update entities through update queries that delete, insert, or modify entities. This section describes how you can update entities with two types of update queries: single-statement updates, which contain one statement, and multiple-statement updates, which enable you to intersperse many attribute assignments among program statements.

Appendix A. Sample Programs

This appendix features complete COBOL74, ALGOL, and Pascal program examples. These examples use concepts described in the guide.

Appendix B. Language Comparison Tables

This appendix features tables that compare concepts explained in this guide with the actual constructs used in COBOL74, ALGOL, and Pascal.

Appendix C. Exception Fields and Categories

This appendix lists and explains the exception fields and categories.

Related Product Information

Unless otherwise stated, all documents referred to in this publication are MCP/AS documents. The titles have been shortened for increased usability and ease of reading.

The following documents are included with the software release documentation and provide general reference information:

- The *Glossary* includes definitions of terms used in this document.
- The *Documentation Road Map* is a pictorial representation of the Product Information (PI) library. You follow paths through the road map based on tasks you want to perform. The paths lead to the documents you need for those tasks. The Road Map is available on the PI Library CD-ROM. If you know what you want to do, but don't know where to find the information, start with the Documentation Road Map.
- The *Information Availability List* (IAL) lists all user documents, online help, and HTML files in the library. The list is sorted by title and by part number.

The following documents provide information that is directly related to the primary subject of this publication.

MCP/AS ALGOL Programming Reference Manual, Volume 2: Product Interfaces

This manual describes the extensions to the Extended ALGOL language that allow application programs to use the Advanced Data Dictionary System (ADDS), the Communications Management System (COMS), the Data Management System II (DMSII), the Screen Design Facility Plus (SDF Plus), or the Semantic Information Manager (SIM). This manual is written for programmers who are familiar with Extended ALGOL programming language concepts and terms.

MCP/AS Binder Programming Reference Manual

This manual describes the functions and applications of the Binder, an efficiency tool that reduces the need to recompile an entire program when only a portion of the program has been modified. This manual is written for programmers who are familiar with programming language concepts and terms.

MCP/AS COBOL ANSI-74 Programming Reference Manual, Volume 2: Product Interfaces

This manual describes the extensions to the standard COBOL ANSI-74 language. These extensions are designed to allow application programs to interface with the Advanced Data Dictionary System (ADDS), the Communications Management System (COMS), the Data Management System II (DMSII), the DMSII Transaction Processing System (TPS), the Screen Design Facility (SDF), the Screen Design Facility Plus (SDF Plus), and Semantic Information Manager (SIM) products. This manual is written for programmers who are familiar with COBOL74 programming language concepts and terms.

MCP/AS InfoExec Capabilities Manual

This manual discusses the capabilities and benefits of the InfoExec data management system. This manual is written for executive and data processing management.

A Series InfoExec DMS.View Operations Guide

This guide explains the InfoExec DMS.View utility and provides operating instructions for it. It is written for Data Management System II (DMSII) users who want to create a Semantic Information Manager (SIM) description and a Structured Query Language Database (SQLDB) description for a DMSII database.

MCP/AS InfoExec Semantic Information Manager (SIM) Technical Overview

This overview describes the SIM concepts on which the InfoExec data management system is based. This overview is written for end users, applications programmers, database designers, and database administrators.

MCP/AS Pascal Programming Reference Manual, Volume 2: Product Interfaces

This manual describes the Pascal interfaces and extensions for the following products: the Advanced Data Dictionary System (ADDS), the Communications Management System (COMS), the Screen Design Facility Plus (SDF Plus), and the Semantic Information Manager (SIM). This manual is written for application programmers who are familiar with Pascal programming language concepts and terms.

Section 1
Programming in the InfoExec Environment

Through the use of statements in COBOL74, ALGOL, and Pascal, you can inquire about and update Semantic Information Manager (SIM) databases in the InfoExec environment. You can also access the data dictionary used to store data items. This guide discusses queries conceptually and explains how and why you use them. (Accessing the data dictionary is described only briefly.) For specific statement syntax and explanations, consult the *ALGOL Reference Manual*, *Vol. 2*, the *COBOL74 Reference Manual*, *Vol. 2*, or the *Pascal Reference Manual*, *Vol. 2* as appropriate.

Query Programming

A query is a statement that inquires about or updates entities in a SIM database. To create and use a query in a program, do the following:

1. Declare the query variable. Query variable declarations name the variables that are temporarily associated with query statements.
2. Declare the query record description. Query record descriptions name the record that will contain data obtained with a SELECT query. They also name the query record fields to be associated with database attributes. Query records are used only in retrieval queries.

   After declaring the query, you can use query statements to do the tasks described in the following steps.
3. Put the program in transaction state, if necessary. Transaction state enables the program to process update statements. These updates are committed to the database only when the program leaves transaction state without aborting the updates.
4. Process the query. Process a query by writing statements that select and retrieve entities and statements that insert, delete, and modify entities in the database.
5. Close the query. When finished with a query, close it either automatically or with the DISCARD statement.

These steps are described in later sections.

You can write queries directly in the main program, or in separately compiled subprograms that are later bound by SYSTEM/BINDER to the main program. You can bind query variables, query records, databases, and entity-reference variables.

Note: *If you bind together several programs that use the same SIM database, then at run time the exception information stored in the DMSTATE exception word is not retained across the bound programs. The first bound program to use the SIM database can retrieve DMSTATE exception information without any problems. However, immediately after a second program that is bound to the first program uses the SIM database, the DMSTATE exception word is reinitialized and the original exception information is no longer available.*

To bind a structure in a subprogram to the same structure in the main program, you declare the structure in the main program. In the subprogram, however, you declare the structure as a global structure. For more information on declaring structures, see the *ALGOL Reference Manual, Vol. 2*, the *COBOL74 Reference Manual, Vol. 2*, or the *Pascal Reference Manual, Vol. 2*.

For more information on binding, see the *Binder Programming Reference Manual.* For more information on global declarations, refer to the appropriate language manual.

Data Dictionary Programming

Other COBOL74, ALGOL, and Pascal extensions enable you to use item entities, record entities, record collection entities, programs, files, form libraries, and databases from the data dictionary in a program. These extensions include the following:

- Entity qualification, which identifies the status of the entities in a program.
- DICTIONARY option, which identifies the attributes of the data dictionary used when a program is compiled. This option also tracks programs stored in the data dictionary.
- Data description entries, which access the data description from the data dictionary.

For more information on these extensions, consult the appropriate language manual.

Section 2
Starting a Query Program

A query program begins with a declaration section that declares a database, identifies query variables and their associated class or query record description, and creates query record fields. The program then opens the database, enabling you to write query statements on the database.

The following program fragment declares the query variables EMP-Q and MNGR-Q, and opens the ORGANIZATION database. This fragment is referred to throughout this section.

```
DATA DIVISION.
DATA-BASE SECTION.
DB PROJEMPDB = "ORGANIZATION" VALUE OF DBKIND IS SEMANTIC.

QD   EMP-Q.
01   EMP-REC.
     02  EMP-NAME.
         03 E-LASTNAME    PIC X(20).
         03 E-FIRSTNAME   PIC X(15).
         03 E-MI          PIC X.
     02  E-SALARY    PIC 9(9)V2.
     02  MNGR-NAME   PIC X(20).
     02  MNGR-ERV USAGE ENTITYREFERENCE OF MANAGER.
QD   MNGR-Q OF MANAGER.

PROCEDURE DIVISION.
BEGIN.
    OPEN UPDATE PROJEMPDB.
```

Declaring a Database

Database declarations identify the name and type of database used in the query. In the program fragment, the following is the database declaration:

```
DATA-BASE SECTION.
DB PROJEMPDB = "ORGANIZATION" VALUE OF DBKIND IS SEMANTIC.
```

This fragment declares a SIM database called ORGANIZATION equating it to the name PROJEMPDB.

For more information on the previous statements, consult the *COBOL74 Reference Manual*, *Vol. 2*. Information on similar ALGOL and Pascal statements can be found in the *ALGOL Reference Manual*, *Vol. 2* and the *Pascal Reference Manual*, *Vol. 2*, respectively.

Declaring a Query

Query declarations identify the query variables and list the classes or query record variable used in the query. There are two kinds of queries: retrieval and update.

Retrieval Queries

Retrieval queries select and retrieve entities but do not modify them. These queries can use the SELECT and RETRIEVE statements but not the update statements. The following program fragment declares a query variable EMP-Q that is used for retrieval queries:

```
QD   EMP-Q.
01   EMP-REC.
     02  EMP-NAME.
         03 E-LASTNAME    PIC X(20).
         03 E-FIRSTNAME   PIC X(15).
         03 E-MI          PIC X.
     02  E-SALARY       PIC 9(9)V2.
     02  MNGR-NAME      PIC X(20).
     02  MNGR-ERV USAGE ENTITYREFERENCE OF MANAGER.
```

Retrieval query declarations contain the name of the query variable and describe a record containing the data to be returned by the query. For more information on retrieval queries, see Section 4, “Retrieving Entities.”

A retrieval query declaration contains both query variables and query record descriptions, which are discussed in the following paragraphs.

Query Variable

The query variable represents the query. Although you can associate one query variable with more than one query statement throughout the program, you can associate a query variable with one statement at a time. Embedded SELECT statements can associate more than one query variable with one query statement.

In the program fragment, the query variable is EMP-Q. Throughout this guide, all query variables end with the letter *Q*.

For more information on the SELECT statement, see “SELECT Statement” in Section 4, “Retrieving Entities.” For information on embedded SELECT statements, see “Structured Formatting” in the same section.

Query Record Descriptions

Query record descriptions contain the name of the record describing the data to be returned by the query. They also contain the names of the record fields to be associated with database attributes. In the following table, the column on the left shows the fields created by the previously mentioned program fragment. Later SELECT and RETRIEVE statements associate the fields with the attributes shown in the column on the right. SELECT and RETRIEVE statements are described in Section 4, "Retrieving Entities."

Field	Database Attribute
E-LASTNAME OF EMP-NAME	LAST-NAME OF NAME
E-FIRSTNAME OF EMP-NAME	FIRST-NAME OF NAME
E-MI OF EMP-NAME	MID-INITIAL OF NAME
E-SALARY	EMPLOYEE-SALARY
MNGR-NAME	LAST-NAME OF NAME OF EMPLOYEE-MANAGER

If a field and an immediate attribute have exactly the same name, later statements can implicitly associate the field and attribute.

The program fragment also declares an entity-reference variable called MNGR-ERV for the MANAGER class. This is a variable that explicitly refers to one entity. Throughout this guide, entity-reference variables end with the suffix *ERV*. Entity-reference variables are described under "Using Entity-Reference Variables" in Section 5, "Designating Transaction State."

Update Queries

Update queries modify, delete, or insert entities with the MODIFY, DELETE, or INSERT statement.

The following fragment declares a query variable called MNGR-Q. You use MNGR-Q primarily for an update query, although you can also use it for a retrieval query.

```
QD   MNGR-Q OF MANAGER.
```

Query declarations used for update queries contain the name of the query, such as MNGR-Q, and the name of the class to be updated, such as MANAGER.

For more information on update query statements, see "Update Statements" in Section 6, "Updating Entities."

Opening and Closing a Database

After declaring a database, you use the OPEN statement to open the database and declare an access mode. This statement appears in the body of the program and not in the declaration section. Two access modes are available:

- Inquiry mode. In this default mode, the program can only read entities in the database. Use this mode if you do not want to update the database.
- Update mode. Use this mode to read and modify entities in the database.

In the program fragment, the following statement opens the database in update mode:

```
OPEN UPDATE PROJEMPDB.
```

Each program can declare a maximum of 47 database, of which 5 can be open simultaneously. The system can have up to 50 open databases.

You can now write query statements and execute them against the database. Once you are finished querying the database, you can close it with the CLOSE statement. This statement closes the database and deactivates all queries relating to that database. The following example closes the database:

```
CLOSE PROJEMPDB.
```

Accessing SIM, DMSII, and LINC II Databases

You can access both SIM and Data Management System II (DMSII) databases in the same program as follows:

- You can update and query DMSII databases only through DMSII applications.
- You can update SIM databases only through SIM applications.
- You can query SIM databases through SIM and DMSII applications.
- You can query DMSII databases, processed by DMS.View, using DMSII, SIM, and SQLDB applications.
- You can update DMSII databases, processed by DMS.View, using DMSII, SIM, and SQLDB applications.

For more information on DMS.View, refer to the *InfoExec DMS.View Operations Guide.*

You can query LINC II databases using SIM and SQLDB applications, if your LINC II database has been processed by LINC.View. For more information on LINC.View, refer to the *InfoExec LINC.View Operations Guide.*

Writing a Query Statement

The query statement either retrieves or updates entities. The following is an example of a query statement that retrieves entities:

```
SELECT EMP-Q FROM EMPLOYEE
      (E-LASTNAME = LAST-NAME OF NAME, E-SALARY = EMPLOYEE-SALARY)
      WHERE EMPLOYEE-SALARY > 20000.
```

A query statement can consist of the following parts:.

- Command, which is that part of the statement that identifies the query operation. In the example, SELECT is the command. Command statements are described in Section 4, "Retrieving Entities," and Section 6, "Updating Entities." The rest of this guide refers to specific commands as *statements*. For example, SELECT is called the SELECT statement.
- Query variable, which represents the query statement. In the example, EMP-Q is the query variable.
- Perspective, which is the primary class from which the query is directed. Any additional classes are viewed in relation to the perspective. All query statements must have at least one perspective and can have multiple perspectives. You explicitly state the perspective using the FROM clause. For more information on multiple perspectives, see "Global Selection Expressions" in Section 3, "Choosing Entities."

  The example explicitly states the EMPLOYEE class as the perspective.
- Target list, which identifies the group of attributes used in the query. This list is enclosed in parentheses. In the example, LAST-NAME and EMPLOYEE-SALARY comprise the target list. Note that the record fields E-LASTNAME and E-SALARY are associated with the attributes in the list.
- Selection expression, which identifies the entities on which the query operates. In the example, *WHERE EMPLOYEE-SALARY > 20000* is the selection expression. Selection expressions are described in Section 3, "Choosing Entities."

Section 3
Choosing Entities

With SIM programming, you choose the entities you need in a program without worrying about how to move from one structure to the next or how to search for a structure. SIM automatically decides how to access entities from the database.

You choose entities in a query statement by using selection expressions. Selection expressions identify the entities on which a query operates. These expressions state a condition that an entity must meet before it is selected by a query statement.

For example, suppose the following data exist:

Employee	**Salary**
Whitmore	29000
Ferris	22000
Sander	22000
Nguyen	29000
Morgan	25000

Assume you want to list employees who are paid more than $25,000. You can use the following selection expression:

```
EMPLOYEE-SALARY > 25000
```

Note that selection expressions identify only the criteria for selecting entities or values. To make the information about the entities available to the program, you must use the RETRIEVE statement. You can use selection expressions with all update and retrieval queries except for the RETRIEVE and DISCARD statements.

This section discusses the two types of selection expressions, the functions used in such expressions, qualification, binding of names, and roles. Examples use the SELECT statement, which is described in Section 4, “Retrieving Entities.”

Types of Selection Expressions

Depending on the kind of data you want, you can use two types of selection expressions: global and local. In addition, global selection expressions can query a single class or multiple classes.

Global Selection Expressions

Global selection expressions apply to an entire query. They identify conditions under which entities from the perspective class or classes are chosen.

With One Perspective

For example, suppose you want to list all employees who make more than $25,000. You could use the previous expression in a global selection expression. In the following example, the global selection expression is *WHERE EMPLOYEE-SALARY > 25000*:

```
SELECT EMP-Q FROM
      (E-LASTNAME = LAST-NAME OF NAME, E-SALARY = EMPLOYEE-SALARY)
       WHERE EMPLOYEE-SALARY > 25000.
```

The expression produces the following table:

Employee	Salary
Whitmore	29000
Nguyen	29000

In addition to relational operators, global selection expressions can use qualification and extended attributes. Qualification and relational operators are explained later in this section. For more information on extended attributes, see the *InfoExec SIM Technical Overview.*

With Multiple Perspectives

Multiple entities, one from each perspective class, are joined together when the values for their joining attributes meet a condition stated by a global selection expression. The next example uses the following declaration:

```
QD   EMP-Q.
01   EMP-Q-REC.
     02   E-NAME      PIC X(20).
     02   MNGR-NAME   PIC X(20).
```

The following is an example of a global selection expression in a query with multiple perspectives:

```
SELECT EMP-Q FROM PROJECT-EMPLOYEE, MANAGER
  (E-NAME = LAST-NAME OF NAME OF PROJECT-EMPLOYEE,
   MNGR-NAME = LAST-NAME OF NAME OF MANAGER)
  WHERE EMPLOYEE-HIRE-DATE OF PROJECT-EMPLOYEE
        = EMPLOYEE-HIRE-DATE OF MANAGER.
```

If a manager and project-employee have the same hire dates, the expression selects both entities. If either entity has a null hire date or if both have hire dates that do not match, neither is selected. An entity name is selected more than once if it matches more than one entity in the other class. For example, if a manager was hired on 1/5/80 and three project-employees were hired on 1/5/80, the manager's name is selected three times. Each selection matches each of the project-employees hired on the same date.

Local Selection Expressions

Local selection expressions affect specific attributes only. They identify conditions under which values for the attribute are chosen. For example, assume you want to list all employees but only salaries that exceed $25,000. You can use the previously designated expression in a local selection expression. In the following example, the local selection expression is *WITH EMPLOYEE-SALARY > 25000*. It applies only to the attribute called LAST_NAME OF NAME:

```
QD   EMP2-Q.
01   EMP2-REC.
     02  EMP-NAME.
         03 E-LASTNAME    PIC X(20).
         03 E-FIRSTNAME   PIC X(15).
         03 E-MI          PIC X.
     02  E-SALARY       PIC 9(9)V2.

SELECT EMP2-Q FROM EMPLOYEE
   (E-LASTNAME = LAST-NAME OF NAME WITH EMPLOYEE-SALARY > 25000,
    E-SALARY = EMPLOYEE-SALARY)
   WHERE LAST-NAME OF NAME NOT EQUAL "SMITH".
```

When you use RETRIEVE statements with statements that contain local selection expressions, each RETRIEVE statement obtains the next entity in the query and then compares the values of the attributes in the entity against the criteria in the local selection expression. If these values do not satisfy the local selection expression, a null value is returned to the program for the attribute that is qualified by the local selection expression instead of the actual value of the attribute. An error exception only occurs when there are no more entities to be retrieved.

The following table illustrates values that could be returned by the local selection expression *WITH EMPLOYEE-SALARY > 25000*:

Employee	Salary
Whitmore	29000
Ferris	–
Sander	–
Nguyen	29000
Morgan	–

Operators and Functions

Selection expressions can use the following types of operators and functions: relational operators, Boolean operators, order functions, aggregate functions, date and time functions, and string expressions. These operators and functions are described in the following paragraphs.

Relational Operators

Relational operators test for relationships between values. They produce values of true, false, or null. (Null values are explained under "Boolean Operators" later in this section.) Unless otherwise stated, you can use the following operators on any kind of attribute:

- *Equal to*. This and *not equal to* are the only operators you can use on compound attributes.
- *Not equal to*. This and *equal to* are the only operators you can use on compound attributes.
- *Greater than*. AFTER is a synonym for this operator. AFTER is most appropriate for attributes involving dates and times.
- *Less than*. BEFORE is a synonym for this operator. BEFORE is most appropriate for attributes involving dates and times.
- *Less than or equal to*.
- *Exists*. This operator uses only one operand. It is true if the operand has a value that is not null. (Null values are discussed under "Boolean Operators" in this section.) The following is an example:

  ```
  WHERE MANAGER-TITLE EXISTS
  ```

  This statement is true only if MANAGER-TITLE has a value that is not null.

You can use relational operators on single-valued and multivalued attributes. Both sides of the operator can be multivalued only if you are using the expression with tabular formatting. Tabular formatting is described under "Entity Formats" in Section 4, "Retrieving Entities."

Boolean Operators

The standard Boolean operators are AND, OR, and NOT. The relational operators, including ISA, have higher precedence than the Boolean operators. Within the Boolean operators, NOT has the highest precedence, followed by AND and then OR. You can use parentheses to control the order of evaluation.

In addition to the usual true and false values, Boolean expressions can produce null (unknown) values. Null values result when SIM cannot determine whether a relational expression is true or false—if, for example, one or both of the operands has an unknown value.

The following describes the results of using true, false, and null values in a relational expression. *T* represents TRUE, *F* represents FALSE, and a question mark (*?*) represents the unknown or null value.

a	b	a AND b	a OR b	NOT a
T	T	T	T	F
T	F	F	T	F
F	T	F	T	T
F	F	F	F	T
T	?	?	T	F
F	?	F	?	T
?	T	?	T	?
?	F	F	?	?
?	?	?	?	?

When testing for TRUE or FALSE, you must take the null value into account. For example, assume the following entities exist in the EMPLOYEE class:

LAST-NAME	BIRTH-DATE
Smith	01/02/55
Jones	?
Martinez	10/27/35

To use all these entities through BIRTH-DATE, you must create an expression that tests for the unknown BIRTH-DATE value as well as for the values 01/02/55 and 10/27/35. The following is an example of such an expression:

```
SELECT EMP-Q FROM EMPLOYEE
WHERE BIRTH-DATE = 01/02/55 OR BIRTH-DATE = 10/27/35
          OR NOT BIRTH-DATE EXISTS.
```

The expression *NOT BIRTH-DATE EXISTS* selects BIRTH-DATE if it is null, that is, if the NOT EXISTS condition is met.

Symbolic Order Functions

Symbolic order functions work only on attributes that are declared with a symbolic order type. Such attributes have an enumerated list of identifiers with a defined order. Symbolic order functions identify the value that precedes or follows another value. For example, the PERSON class contains the compound attribute EDUCATION, which consists of the symbolic ordered type DEGREE-OBTAINED. DEGREE-OBTAINED can have the following values: HS, BA, BS, MA, MS, PHD. (Compound attributes are described under "ASSIGN Statement with Compound Attributes," and "INCLUDE Statement with Compound Attributes" in Section 6, "Updating Entities."

You can find out what degree comes before or after another through the following operators:

- Predecessor. This operator returns the value preceding the operand in a list. You cannot use it on the first value in a list. To find the value preceding MA (Master of Arts) use the following expression:

  ```
  PRED(MA)
  ```

  This expression returns the value BS (Bachelor of Science).

- Successor. This operator returns the value succeeding the operand in a list. You cannot use it on the last value of a list. To find the entity after MA, use the following expression:

  ```
  SUCC(MA)
  ```

  This expression returns the value MS (Master of Science).

Aggregate Functions

Aggregate functions apply to a collection of values and produce one value. Such functions can contain a local selection expression to select the collection of values to which the function is applied.

The following are known as aggregate functions:

- Average. This function calculates the average of a collection of numeric values. Null values are not counted in the average. The following example averages manager bonuses in the Attractions department. Unknown bonuses are not included.

```
AVG(BONUS OF MANAGER)
WHERE DEPT-TITLE OF MANAGERS-DEPT = "ATTRACTIONS"
```

- Count. This function applies to both classes and attributes. This is the only function you can use with entity valued attributes.

  When used with a class, this function counts the number of entities in the class. When used with multivalued attributes, this function counts the number of values that are not null. When used with single-valued attributes, this function equals 0 (zero) if the attribute has a null value and 1 if it has no null values. The following example counts the number of employees managed by Jones:

```
COUNT(EMPLOYEES-MANAGING) OF MANAGER
WHERE LAST-NAME OF NAME = "JONES"
```

- Minimum. This function selects the minimum value from a collection of values. It does not include null values. The following example selects the lowest manager bonus:

```
MIN(BONUS OF MANAGER)
```

- Maximum. This function selects the maximum value from a collection of values. It does not include null values. The following example selects the highest manager bonus:

```
MAX(BONUS OF MANAGER)
```

- Sum. This function totals all the numeric values in a collection. It ignores null values. The following example totals all manager bonuses:

```
SUM(BONUS OF MANAGER)
```

The location of the parentheses in aggregate functions identifies the particular values used in the collection. Qualifying the attributes within the parentheses gives one value for all the entities in the perspective. For example, the following statement counts all the employees being managed in the database. It produces one value.

```
COUNT(EMPLOYEES-MANAGING OF MANAGER)
```

Qualifying the attributes outside the parentheses gives a value for each entity in the perspective. For example, the following statement counts the employees for each manager. It produces a value for each manager in the database.

```
COUNT(EMPLOYEES-MANAGING) OF MANAGER
```

Note that if there are no entities in the class on which you are using the aggregate function, the COUNT function produces zero while the MIN and MAX functions produce null values.

For more information on aggregate functions, refer to the *SIM OML Programming Guide*.

Date and Time Functions

In SIM, special data types represent the date and time. Each type has a specific format for representing constant values, and several functions that operate on or produce values of that type. For more information on date and time functions, see the *ALGOL Reference Manual*, *Vol. 2*, the *COBOL74 Reference Manual*, *Vol. 2*, or the *Pascal Reference Manual*, *Vol. 2*.

String Expressions

String expressions produce string values or consist of operands that are strings. For example, to join two or more strings, use the CAT operator. The following is an example:

```
"ABC" CAT "DEF" = "ABCDEF"
```

Pattern Matching

Pattern matching can be done with the following ISIN operator. This operator is true if the string conforms to the pattern. The following is a true expression:

```
"John Smith" ISIN {A..Za..z }?
```

This expression is true because all the characters in the string *John Smith* satisfy the pattern *{A..Za..Z}?*

Pattern matching can use special symbols to control the match. In the previous example, the braces ({}) are special symbols. Table 3–1 describes these special symbols and what they mean. It also gives an example of a true expression using the symbol.

Table 3–1. Symbols for Matching Patterns in String Expressions

Symbol	Meaning	Example
_	Used to match any single character.	NAME ISIN L_nd This example returns a true value if NAME includes a value such as Lind, Lund, or Lynd.
*	Used to match zero or more occurrences of the preceding character in the statement.	NAME ISIN Phil* This example returns a true value if NAME includes a value such as Phi, Phil, or Phill.
{}	Used to match any character given within the braces.	FIRST-NAME ISIN {BJG}ill This example returns a true value if FIRST-NAME includes a value such as Bill, Jill, or Gill.
I	Used to match the pattern on either side of bar.	TITLE ISIN (ts I ar cz)ar This example returns a true value if TITLE includes a value such as tsar, or czar.
()	Used to group expressions.	PROJECT-NUMBER ISIN (12 I 34)5* This example returns a true value if PROJECT-NUMBER includes a value such as 125, 345, 1255, or 12.
""	Used when symbols are included in the string expression.	TITLE ISIN XX"_"YY This example returns a true value if TITLE has a value XX_YY.
?	Used as a shorthand for _*, that is, to denote a repeat of any character.	NAME ISIN L?ND This example returns a true value if NAME has values such as LOOND, or LOND.
..	Used with braces to denote an allowable range of characters.	NAME ISIN L{s..z}nd This example returns a true value if NAME has a value such as Lund, or Lynd.

Note that a pattern ends with a blank. To make blanks part of a pattern, you must enclose the blank within quotation marks (").

Note also that to use a quotation mark as part of a string, you must use two quotation marks ("").For example, the string *A"B* must be expressed as *A""B*.

Comparing Strings

You can also compare strings using any of the previously mentioned relational operators such as *less than* or *greater than*. Two strings are equal if they are the same length and every character in the first string matches every character in the second string.

If two strings are not equal, then the first nonmatching character, from left to right, determines their relative ordering. That character's EBCDIC value is used in the comparison. If the character in the first string precedes the corresponding character in the second string, the first string is less than the second. Otherwise, the second string is less than the first.

If an operator compares two strings of different lengths, SIM extends the shorter operator with blank characters at the end.

The following sample comparisons are true:

```
"abc" < "ABC"
"A" < "Z"
"1" > "Zed"
```

SIM also has the following functions for strings:

- Length. This function counts the number of characters in a designated string. The following is an example:

  ```
  LEN("SMITH") = 5
  ```

- Extract. This function extracts a substring from a string. The following is an example:

  ```
  EXT("JONATHAN",1,3) = "JON"
  ```

- Position. This function returns the starting position of a substring within a string at or after the position identified by an integer. If the integer is omitted, the function assumes 1 for the integer. If the substring is not found in the string, the function returns zero. The following is an example:

  ```
  POS("R","BARRY",4) = 4
  ```

  In this case, the function returns *4* because that is the position at or after the designated integer (4) at which it finds *R*.

- Repeat . This function returns a string, repeated a designated number of times. The following is an example:

  ```
  RPT("JUDY ",3) = "JUDY JUDY JUDY "
  ```

- Character. This function constructs a string using EBCDIC characters represented by the hexadecimal numbers in the function. It is most useful for including nonprinting characters in a string. The following is an example:

  ```
  4"C296A9" = "Boz"
  ```

Qualification

Qualification ensures that attributes in a target list clearly relate to a specific class or that a class clearly relates to a specific database. For more information on qualification, refer to the *SIM OML Programming Guide*.

Qualification rules differ from language to language. For more information on qualification, refer to the appropriate language manual. The following paragraphs describe qualification as it applies to all languages.

Factoring

To save entry time, you can factor out common elements in a list of attributes. For example, assume you have the following list:

```
LAST-NAME OF NAME OF PERSON, BIRTH-DATE OF PERSON, AGE OF PERSON
```

You can factor out the common element *OF PERSON* and enter it only once, as in the following example:

```
(LAST-NAME OF NAME, BIRTH-DATE, AGE) OF PERSON
```

Note that you can only use factoring in the target list of a SELECT statement.

INVERSE Function

In some cases, although an attribute exists from one class to another class, no inverse attribute exists between the two classes. You can use the INVERSE function to create such an attribute if an implicit or explicit inverse exists between the classes. Databases converted from DMSII to SIM frequently do not have implicit or explicit inverses between classes. For more information on converting databases, see the *InfoExec DMS.View Operations Guide*.

For example, the following diagram shows that the DEPT-ASSIGNED attribute exists from the PROJECT class to the DEPARTMENT class. However, there is no corresponding inverse attribute from DEPARTMENT to PROJECT.

```
PROJECT ---------------------------- > DEPARTMENT
              (DEPT-ASSIGNED)
```

You can temporarily designate an inverse attribute from DEPARTMENT to PROJECT by using the INVERSE function. For example, the function INVERSE OF DEPT-ASSIGNED accesses PROJECT attributes through the DEPARTMENT class. The next example uses the following declaration:

```
QD DEPT-Q.
01 DEPT-Q-REC.
   02  TITLE   PIC X(20).
```

The following example uses the INVERSE function in a query expression:

```
SELECT DEPT-Q FROM DEPARTMENT
   (TITLE = PROJECT-TITLE OF INVERSE OF DEPT-ASSIGNED)
 WHERE DEPT-TITLE = "ATTRACTIONS".
```

The example chooses the titles of projects in the ATTRACTIONS department.

Binding of Names

A statement uses the same value for a class or an attribute anywhere that class or attribute appears in a statement unless a construct is used to change the value. This rule is known as binding of names.

For example, if the LAST-NAME of MANAGER equals JONES at the beginning of a statement, that LAST-NAME still equals JONES at the end of the statement. For more information on binding of names, refer to the *SIM OML Programming Guide*.

The binding of names is best illustrated by reference variables. When a name refers to a collection of objects such as a class, an entity-valued attribute, or a multivalued attribute, the name establishes an implicit reference variable. This variable contains each object in the collection as the query is processed.

The next example uses the following declaration:

```
QD MNGR-Q.
01 MNGR-Q-REC.
   02   MNGR-NAME PIC X(20).
   02   STAFF-NAMEPIC X(20).
```

The following is an example of name binding in a query:

```
SELECT MNGR-Q FROM MANAGER
  (MNGR-NAME = LAST-NAME OF NAME OF MANAGER,
   STAFF-NAME = LAST-NAME OF NAME OF EMPLOYEES-MANAGING OF MANAGER)
  WHERE LAST-NAME OF NAME OF MANAGER = "JONES".
```

The query examines each manager named Jones to select his staff. As the query examines each entity in the MANAGER class, it successively puts each entity (each manager) into a single reference variable. Although the query refers to MANAGER three times *(LAST-NAME OF MANAGER*, *LAST-NAME OF EMPLOYEES-MANAGING OF MANAGER*, *and WHERE LAST-NAME OF MANAGER = "JONES")*, all references refer to the same manager.

In some cases, you might want to use different class or attribute values in the same statement. In other words, you might want to use two or more reference variables to the same class by *breaking* the binding. For example, to use the manager whose LAST-NAME equals JONES and another manager whose LAST-NAME equals SMITH, use a reference variable for the entity Jones and another variable for the entity Smith.

You can break the binding in a query by using the CALLED clause, quantifiers, certain functions, or path expressions. These constructs are described in the following paragraphs and use the following table of attributes from the MANAGER class perspective:

LAST-NAME	EMPLOYEE-SALARY	BONUS
Smith	23000	1000
Jones	21000	1000
Chen	18000	2000
Roberts	24000	2000

CALLED Clause

This clause explicitly assigns a reference variable to an entity accessed through a class or multivalued attribute. The CALLED clause also names the variable, enabling you to use the variable subsequently within the same statement.

In the following example, the query explicitly assigns a reference variable called POORER to an entity with an EMPLOYEE-SALARY of under $20,000. The query also implicitly assigns a reference variable to an entity with an EMPLOYEE-SALARY of over $22,000.

```
SELECT MNGR-Q FROM MANAGER
   WHERE (EMPLOYEE-SALARY OF EMPLOYEES-MANAGING > 22000)
   AND (EMPLOYEE-SALARY OF EMPLOYEES-MANAGING CALLED POORER < 20000).
```

The POORER reference variable is assigned Chen whose salary is $18,000 *(EMPLOYEE-SALARY < 20000)*. The implicit and unnamed reference variable is assigned both Smith, whose salary is $23,000 *(EMPLOYEE-SALARY > 22000)*, and Roberts, whose salary is $24,000.

The CALLED clause can follow a class name in the FROM clause. A CALLED clause cannot follow a variable name that was declared by another CALLED clause.

Quantifiers

You can also create implicit reference variables by stating whether some, all, or no values of a multivalued attribute must meet a condition. SOME, ALL, and NO are quantifiers that can be used only in selection expressions. A SOME condition is true if at least one entity meets the condition. An ALL condition is true if all entities meet the condition. A NO condition is true if none of the entities meet the condition.

In the following example, the query assigns an implicit reference variable to a set of entities whose EMPLOYEE-SALARY is over $22,000 and another reference variable to a set of entities whose EMPLOYEE-SALARY value is under $20,000.

```
SELECT MNGR-Q FROM MANAGER
   WHERE SOME (EMPLOYEE-SALARY OF EMPLOYEES-MANAGING) > 22000
   AND SOME (EMPLOYEE-SALARY OF EMPLOYEES-MANAGING) < 20000.
```

The first SOME condition assigns an implicit variable to the entity Smith, whose salary is $23,000 *(EMPLOYEE-SALARY > 22000)*, and to the entity Roberts, whose salary is $24,000. The second SOME condition assigns an implicit variable to the entity Chen, whose salary is $18,000 *(EMPLOYEE-SALARY < 20000)*.

Because the variables created by quantifiers are not named, you cannot use these variables elsewhere in the program. Reference variables created by the CALLED function cannot use quantifiers.

Functions

A class or a multivalued attribute appearing in the argument of an aggregate function creates a new reference variable. This variable is separate from the other appearance of the class or attribute in the query.

In the following example, the function AVG(BONUS OF MANAGER) creates a new reference variable that is separate from the reference variable created for LAST-NAME and BONUS.

```
SELECT MNGR-Q FROM MANAGER
   (MNGR-NAME = LAST-NAME OF NAME)
   WHERE BONUS > AVG(BONUS OF MANAGER).
```

Aggregate functions are described under “Aggregate Functions” in this section.

Subclass and Superclass Roles

An entity can play different roles in a SIM database. For example, assume Jones is an employee who is an interim manager. Thus, he is an entity in the superclass EMPLOYEE. He also has roles in the subclasses PROJECT-EMPLOYEE, INTERIM-MANAGER, and MANAGER.

To determine if an entity plays a certain role in a class, use the ISA operator. To access an entity from a specific class, use the AS clause.

Roles are further described in the *SIM OML Programming Guide*.

ISA Operator

The ISA operator tests whether an entity plays a certain role. The first operand must be either a class name or entity-valued attribute. The second operand must be a class name. An expression using this operator is true only if the entity in the first operand is a member of the class in the second operand. The following is an example:

```
SELECT EMP-Q FROM EMPLOYEE
  (E-NAME = LAST-NAME OF NAME)
  WHERE SPOUSE ISA MANAGER.
```

This query selects employees only if their spouses are entities in the MANAGER class.

The ISA operator is further described in the *SIM OML Programming Guide*.

AS Clause

The AS clause qualifies an entity by identifying the role from which it is accessed. This clause designates the subclass of an entity if the query's perspective is a superclass. The following is an example:

```
SELECT ORG-Q FROM PERSON
  (E-NAME = LAST-NAME OF NAME OF SPOUSE AS MANAGER)
  WHERE BONUS OF PERSON AS MANAGER = 1000.
```

The perspective of the query is the PERSON superclass. To get the names of spouses who are managers, you access LAST-NAME OF SPOUSE from the MANAGER subclass using the clause AS MANAGER. Note that if there are no spouses who are managers, E-NAME assumes a null value.

The AS clause is further described in the *SIM OML Programming Guide*.

Section 4
Retrieving Entities

Entities are retrieved through retrieval queries, which consist of the SELECT and RETRIEVE statements. This section describes how to use those statements as well as how to format entity values and recursively select attributes through transitive closure.

SELECT Statement

The SELECT statement selects the entities you want to use. However, it does not make the information about those entities available to the program. The RETRIEVE statement does. The SELECT statement must have a query variable and a perspective. It can optionally have a target list and a global selection expression. The following is an example of a SELECT statement:

```
SELECT EMP-Q FROM PROJECT-EMPLOYEE
  (E-NAME = LAST-NAME OF NAME, E-SALARY = EMPLOYEE-SALARY)
  WHERE EMPLOYEE-SALARY > 20000.
```

The perspective class is PROJECT-EMPLOYEE and the target list comprises LAST-NAME and EMPLOYEE-SALARY. The global selection expression is *WHERE EMPLOYEE-SALARY > 20000*.

The SELECT statement also activates a retrieval query, thus enabling it to be processed by the system. The query, as represented by the query variable, remains active until you use the DISCARD statement on the query variable, or until you use a SELECT statement on an active query. If the query was activated within transaction state, it can also be deactivated by the statement that ends the transaction.

For more information on parts of a SELECT statement, see “Writing a Query Statement” in Section 2, “Starting a Query Program.” For more information on the DISCARD statement, see “DISCARD Statement” later in this section. For more information on transaction state, see Section 5, “Designating Transaction State.”

Assigning Entity Values to Fields

A query declaration can create query record fields, as shown in "Declaring a Query" in Section 2, "Starting a Query Program." The SELECT statement associates these fields with database attributes. SIM then assigns the values of these attributes to the fields, enabling their use in expressions.

For example, assume you previously declared a query variable called EMP-Q with the query record fields E-NAME, E-SALARY, and MNGR-NAME. The following statement associates those fields with their corresponding database attributes through the EMPLOYEE class perspective:

```
SELECT EMP-Q FROM EMPLOYEE
     (E-NAME = LAST-NAME OF NAME,
      E-SALARY = EMPLOYEE-SALARY,
      MNGR-NAME = LAST-NAME OF NAME OF EMPLOYEE-MANAGER).
```

Alternatively, as in the following example, you can associate the fields with attributes seen through the MANAGER class perspective. The attributes of EMPLOYEE become attributes of the entity-valued attribute EMPLOYEES-MANAGING.

```
SELECT EMP-Q FROM MANAGER
     (E-NAME = LAST-NAME OF NAME OF EMPLOYEES-MANAGING,
      E-SALARY = SALARY OF EMPLOYEES-MANAGING,
      MNGR-NAME = LAST-NAME OF NAME).
```

Note that you can use the same query variable in subsequent SELECT statements, if you deactivate the variable before using it again. This process is further explained under "DISCARD Statement" later in this section.

Attributes in the Target List

If your record field names are the same as the attribute names, you need not list all the attributes in the target list. You can list some or none of the attributes, and SIM automatically gets the values of the unlisted attributes.

The next example uses the following declaration:

```
QD    MNGR-Q.
01    MANAGER-REC.
      02   LAST-NAME        PIC X(20).
      02   PERSON-ID        PIC 9(9).
      02   EMPLOYEE-SALARY  PIC 9(5).
```

In the following example, only LAST-NAME and PERSON-ID are identified in the target list. Nevertheless, SIM automatically gets the value of the implicit attribute EMPLOYEE-SALARY.

```
SELECT MNGR-Q FROM MANAGER
     (LAST-NAME = LAST-NAME OF NAME, PERSON-ID = PERSON-ID).
```

Sorting Attribute Values

You can designate the attribute by which your target list is sorted and indicate ascending or descending order. Do this with the ORDER BY clause as in the following example:

```
SELECT EMP-Q FROM EMPLOYEE
  (E-NAME = LAST-NAME OF NAME, E-SALARY = EMPLOYEE-SALARY)
  ORDER BY ASCENDING LAST-NAME OF NAME
  WHERE EMPLOYEE-SALARY > 20000.
```

In this example, names and salaries are sorted by last name in ascending order.

Removing Duplicate Data

You can select unique data and remove any duplicates through the DISTINCT clause. In the following example, only unique data in the EMPLOYEE class is selected:

```
SELECT EMP-Q FROM EMPLOYEE DISTINCT
   (E-NAME = LAST-NAME OF NAME, E-SALARY = EMPLOYEE-SALARY)
   WHERE EMPLOYEE-SALARY > 20000.
```

You can use the DISTINCT clause only with tabular formatting. Tabular formatting is described under “Tabular Formatting” later in this section.

RETRIEVE Statement

As stated previously, the SELECT statement only selects the entities for a query. The RETRIEVE statement assigns the selected entity values to the program variables by retrieving the query variable associated with those entities. For more information on query variables, see "Query Variable" in Section 2, "Starting a Query Program."

For example, to retrieve entity values set up by a previous SELECT statement that included the PROJ-Q query variable, use the following statement:

```
RETRIEVE PROJ-Q.
```

Effect of Target Expressions on Retrieval Data

The attributes and expressions in the target list of a SELECT statement, referred to as *target expressions*, affect the values returned from a RETRIEVAL statement. The retrieval data varies depending on the following:

- The number of single-valued or multivalued target expressions included in the SELECT statement
- The relationship between the target expressions in the SELECT statement

Effect of Single-Valued Target Expressions

A target expression in a SELECT statement is considered to be single-valued under one of the following conditions:

- The target expression is a single-valued attribute that is immediate to or inherited by the perspective class.

  In the following example, PROJECT_NO is a single-valued target expression because it is an immediate, single-valued attribute of the perspective class PROJECT:

  ```
  SELECT PROJ-Q FROM PROJECT
     (PROJECT_NO).
  ```

- The target expression is an extended, single-valued attribute for which the entity-valued attribute (EVA) connecting the extended attribute to the perspective class is single-valued.

  In the following example, PROJECT_TITLE OF PROJECT_OF is a single-valued target expression because PROJECT_TITLE is a single-valued, extended attribute that is connected to the perspective class ASSIGNMENT through the single-valued EVA *PROJECT_OF*:

  ```
  SELECT PROJ-Q FROM ASSIGNMENT
     (ASSIGNMENT_NO, PROJECT_TITLE OF PROJECT_OF).
  ```

For queries that include single-valued target expressions, each execution of the RETRIEVAL statement returns the data in one record until no more data exists to return.

Assume you want to retrieve the project number and the name of a project. You would write the query as follows:

```
SELECT PROJ-Q FROM PROJECT
   (PROJECT_NO,PROJECT_TITLE).
```

PROJECT_TITLE and PROJECT_NO are single-valued target expressions because they are single-valued attributes in the perspective class PROJECT.

The following example shows the records of data returned from this query:

```
101      Camelot
102      Excalibur
103      Gallahad
201      Camelot1
202      Excalibur1
203      Gallahad1
```

Effect of Multivalued Target Expressions

A target expression in a SELECT statement is considered to be multivalued under one of the following conditions:

- The target expression is a multivalued attribute or a part of a multivalued compound attribute that is immediate to or inherited by the perspective class.

  In the following example, GPA OF EDUCATION is a multivalued target expression because GPA is part of the immediate compound attribute EDUCATION:

  ```
  SELECT PERSON-Q FROM PERSON
     (LAST_NAME OF NAME,GPA OF EDUCATION).
  ```

- The target expression is a multivalued extended attribute or is connected to the perspective class by a multivalued EVA.

  In the following example, although PROJECT_NO is a single-valued attribute, it is a multivalued target expression because it is connected to the perspective class PROJECT_EMPLOYEE by the multivalued EVA *CURRENT_PROJECT*:

  ```
  SELECT EMP-Q FROM PROJECT_EMPLOYEE
     (LAST_NAME OF NAME,PROJECT_NO OF CURRENT_PROJECT).
  ```

- The target expression is in an embedded SELECT statement that is connected to the previous SELECT statement by a multivalued EVA.

  In the following example, although PROJECT_NO is a single-valued attribute, it is a multivalued target expression because it is connected to the perspective class PROJECT in the previous SELECT statement by the multivalued EVA *CURRENT_PROJECT*:

```
SELECT PROJ_Q FROM PROJECT
   (PROJECT_TITLE)
    SELECT EMP-Q FROM PROJECT_EMPLOYEE
       (LAST_NAME OF NAME,PROJECT_NO OF CURRENT_PROJECT).
```

Two multivalued target expressions in a SELECT statement can be either dependent on or independent of each other based on the following conditions:

- The target expressions are dependent if both are in the same compound attribute or if both are single-valued, extended attributes in the same class and are connected to the perspective class by the same multivalued EVA.

  In the following example, DEGREE_OBTAINED OF EDUCATION and GPA OF EDUCATION are dependent because DEGREE_OBTAINED and GPA are part of the same compound attribute EDUCATION. In addition, PROJECT_NO and PROJECT_TITLE are dependent on each other because both are single-valued extended attributes from the class PROJECT and both are connected to the perspective class PROJECT_EMPLOYEE by the multivalued EVA *CURRENT_PROJECT*.

```
SELECT LAST NAME OF NAME FROM PROJECT_EMPLOYEE
   (DEGREE_OBTAINED OF EDUCATION, GPA OF EDUCATION,
   PROJECT_NO OF CURRENT PROJECT, PROJECT_TITLE OF CURRENT PROJECT).
```

- The target expressions are independent if either of the two target expressions is declared as a multivalued attribute or if they are connected to the perspective class by different multivalued EVAs.

  In the following example, PROJECT_TITLE and LAST_NAME are independent because they are connected to the perspective class MANAGER by two different multivalued EVAs, *PROJECTS_MANAGING* and *EMPLOYEES_MANAGING*:

```
SELECT MANAGER_TITLE FROM MANAGER
   (PROJECT_TITLE OF PROJECTS_MANAGING,
    LAST_NAME OF NAME OF EMPLOYEES_MANAGING).
```

For queries that include multivalued target expressions, each execution of the RETRIEVAL statement returns the data as follows until no more data exists to return:

- The value of only one multivalued expression is returned in a record.
- The values of two or more dependent multivalued expressions are returned in the same record. Subsequent RETRIEVAL statements return the next occurrence of the dependent expressions until none exist.
- The values of two or more independent multivalued target expressions are returned in separate records.

  That is, each RETRIEVE statement returns a value of the first expression and a blank for the second expression until no value for the first expression exists. Then, each RETRIEVE statement returns a blank for the first expression and a value for the second expression until no values for the second expression exist, and so on until no more data exists to return.

Each RETRIEVE statement returns the value of a single-valued target expression for each occurrence of a multivalued target expression. Blank values are displayed for all other occurrences of multivalued target expressions.

The following examples illustrate these effects.

Example of One Multivalued Target Expression

The following query includes the multivalued target expression EMPLOYEE_SALARY OF EMPLOYEES_MANAGING OF PROJECT_MANAGER. In this case, EMPLOYEE_SALARY is a single-valued attribute extended from the class EMPLOYEE, which is connected to the perspective class PROJECT. The connection is through the multivalued EVA *EMPLOYEES_MANAGING* by way of the single-valued EVA *PROJECT_MANAGER*.

```
SELECT PROJ-Q FROM PROJECT
   (PROJECT_TITLE, MANAGER_TITLE OF PROJECT_MANAGER,
   EMPLOYEE_SALARY OF EMPLOYEES_MANAGING OF PROJECT_MANAGER).
```

The following example shows the records of data returned from this query. The query returns the values of the single-valued target expressions PROJECT_TITLE and MANAGER_TITLE once for each value in the multivalued target expression EMPLOYEE_SALARY.

```
Excalibur      Supervisor      15000
Excalibur      Supervisor      17500
Excalibur      Supervisor      14000
Excalibur      Supervisor      20000
Camelot        Executive       50000
Camelot        Executive       60000
Camelot        Executive       45000
```

Example of Two Dependent Multivalued Target Expressions

The following query uses two dependent multivalued target expressions: PROJECT_NO OF CURRENT_PROJECT and PROJECT_TITLE OF CURRENT_PROJECT.

Although PROJECT_NO and PROJECT_TITLE are single-valued attributes of the extended class PROJECT, they become multivalued target expressions because PROJECT is connected to the perspective class PROJECT_EMPLOYEE through the multivalued EVA *CURRENT_PROJECT*.

These two target expressions are dependent on each other because both are single-valued extended attributes from the class PROJECT and both are connected to the perspective class PROJECT_EMPLOYEE by the multivalued EVA *CURRENT_PROJECT*.

```
SELECT EMP-Q FROM PROJECT_EMPLOYEE
   (LAST_NAME OF NAME, PROJECT_NO OF CURRENT_PROJECT,
    PROJECT_TITLE OF CURRENT_PROJECT).
```

The following example shows the records of data returned from this query. The query returns each value of the single-valued target expression LAST_NAME for each pair of values in the dependent multivalued target expression PROJECT_NUMBER and PROJECT_NAME.

```
Carlin         101      Camelot
Carlin         202      Excalibur1
Aquino         102      Excalibur
Aquino         202      Excalibur1
Reinholtz      202      Excalibur1
Reinholtz      103      Gallahad
Reinholtz      203      Gallahad1
```

Example of Two Independent Multivalued Target Expressions

The following query contains these multivalued target expressions:

- PROJECT_TITLE OF PROJECTS_MANAGING.

  This target expression is multivalued because it is an attribute connected to the perspective class MANAGER by a multivalued EVA *PROJECTS_MANAGING*.

- LAST_NAME OF NAME.

  This target expression is multivalued because it is an attribute connected to the perspective class MANAGER by a multivalued EVA *EMPLOYEES_MANAGING*.

These two target expressions are independent of each other because they are connected to the perspective class by different EVAs.

The query is as follows:

```
SELECT MANG-Q FROM MANAGER
   (MANAGER_TITLE, PROJECT_TITLE OF PROJECT_MANAGING,
    LAST-NAME OF NAME OF EMPLOYEES_MANAGING).
```

The following example shows the records of data returned from this query. The query returns each value of the independent multivalued target expressions PROJECT_TITLE and LAST_NAME in separate records. Each value of the single-valued target expression MANAGER_TITLE is repeated once for each value of the multivalued target expressions.

```
Dept_Manager      Excalibur
Dept_Manager      Excalibur2
Dept_Manager                    Feverman
Dept_Manager                    Smythe
Dept_Manager                    Roget
Supervisor        Camelot
Supervisor        Camelot1
Supervisor                      Carrey
Supervisor                      Lani
Supervisor                      Crawford
Supervisor                      Sitar
```

Effect of Multiple Perspective Classes on Retrievals

If the SELECT statement includes more than one perspective class, the subsequent RETRIEVAL statements return each piece of data several times. The data repetition occurs because a matrix is formed for the data retrieval. The requested items are retrieved from every possible combination of entities in the perspective classes, as long as that combination meets the global selection expression criteria. If there is no global selection expression, then the query returns a complete matrix of entities.

Assume you want to retrieve the manager title and project employee title. You would write the query as follows:

```
SELECT MANG-Q FROM MANAGER, PROJECT_EMPLOYEE
   (MANAGER_TITLE OF MANAGER, TITLE OF PROJECT_EMPLOYEE)
WHERE EMPLOYEE_SALARY OF MANAGER = EMPLOYEE_SALARY OF PROJECT_EMPLOYEE.
```

The following example shows the records of data returned from the query:

```
Dept_Manager      Staff
Dept_Manager      Specialist
Dept_Manager      Junior
Div_Manager       Staff
Div_Manager       Specialist
Div_Manager       Junior
Executive         Staff
Executive         Specialist
Executive         Junior
Supervisor        Staff
Supervisor        Specialist
Supervisor        Junior
Dept_Manager      Staff
Div_Manager       Staff
Executive         Staff
Supervisor        Staff
Dept_Manager      Specialist
Div_Manager       Specialist
Executive         Specialist
Supervisor        Specialist
Dept_Manager      Junior
Div_Manager       Junior
Executive         Junior
Supervisor        Junior
```

DISCARD Statement

The DISCARD statement discards a query variable, thus deactivating its associated query. The query can be a retrieval query or an update query. The following example discards a retrieval query called PROJ-Q:

```
DISCARD PROJ-Q.
```

You can now reuse the same query variable with another query.

You need to deactivate queries when you are finished with them so SIM can use resources such as memory more efficiently. For example, the same query variable can be used in several SELECT queries only if each query is closed before the next one is opened. If a SELECT statement tries to activate an already active query, that query is implicitly deactivated before being activated.

Formatting the Output from Retrieval Queries

You can tailor your view of the database to suit the needs of the program logic by formatting the output of a retrieval query. The retrieval query output can be formatted in structured, tabular, or hybrid formats.

For information on how target expressions in the SELECT statement might affect your output, refer to “Effect of Target Expressions on Retrieval Data” earlier in this section.

Structured Formatting

Think of a query as a tree. The base of the tree is the perspective class of the query. As SIM processes the query, it follows and crosses the branches, which are the entity-valued attributes between classes. The intersections between branches and the leaves represent classes. SIM ends processing at one of the branches or leaves.

For example, assume the following structure exists:

```
DEPARTMENT [DEPT-TITLE]
  \
   \  (DEPT-IN)
    \
   PROJECT-EMPLOYEE [LAST-NAME, EMPLOYEE-SALARY]
      \
       \  (PROJECT-TEAM)
        \
      PROJECT [PROJECT-TITLE, PROJECT-NO]
```

PROJECT, PROJECT-EMPLOYEE, and DEPARTMENT are classes. The PROJECT class is the perspective class. PROJECT-TITLE, PROJECT-NO, and PROJECT-TEAM are the immediate attributes of PROJECT.

Structured selection retrieves data in the structure implied by the class relationships used in a query. To retrieve data in a structure, embed SELECT statements within other SELECT statements.

The next example uses the following declaration:

```
QD   PROJ-Q.
01   PROJ-REC.
     02   PF1   PIC X(20).
     02   PF2   PIC 9(11)  BINARY.
QD   EMP-Q.
01   EMP-REC.
     02   PF3   PIC X(20).
     02   PF4   REAL.
QD   DEPT-Q.
01   DEPT-REC.
     02   PF5   PIC X(20).
```

The following example of structured selection shows a query with embedded SELECT statements:

```
SELECT PROJ-Q FROM PROJECT
   (PF1 = PROJECT-TITLE, PF2 = PROJECT-NO
    SELECT EMP-Q FROM PROJECT-TEAM WITH EMPLOYEE-SALARY > 17000
       (PF3 = LAST-NAME OF NAME,
        PF4 = EMPLOYEE-SALARY
        SELECT DEPT-Q FROM DEPT-IN
         WITH DEPT-TITLE NOT EQUAL "ACCOUNTING"
           (PF5 = DEPT-TITLE)))
WHERE PROJECT-NO = 500.
```

Using the RETRIEVE statements on the left produces the data on the right:

Statement	Data
RETRIEVE PROJ-Q.	Preparations 500
RETRIEVE EMP-Q.	Jones 19000
RETRIEVE DEPT-Q.	Food Services
RETRIEVE EMP-Q.	Smith 22000
RETRIEVE DEPT-Q.	Merchandising
RETRIEVE EMP-Q.	Martinez 18000
RETRIEVE DEPT-Q.	Attractions

Tabular Formatting

Tabular formatting organizes data into a table. Each row of the table represents a record. To create a tabular format, write the statement without embedding SELECT statements. The next example uses the following declaration:

```
QD   PROJ-Q.
01   PROJ-REC.
     02   PROJECT-TITLE       PIC X(20).
     02   PROJECT-NO          PIC 9(11)   BINARY.
     02   LAST-NAME           PIC X(20).
     02   EMPLOYEE-SALARY                 REAL.
     02   DEPT-TITLE          PIC X(20).
```

The following is an example of tabular formatting:

```
SELECT PROJ-Q FROM PROJECT
       (PROJECT-TITLE, PROJECT-NO,
       (LAST-NAME OF NAME, EMPLOYEE-SALARY, DEPT-TITLE OF DEPT-IN)
        OF PROJECT-TEAM)
  WHERE PROJECT-NO = 500.
```

Using the RETRIEVE statement on the left produces the data on the right:

Statement	Data
RETRIEVE PROJ-Q.	Preparations 500 Jones 19000 Food Service
RETRIEVE PROJ-Q.	Preparations 500 Smith 22000 Merchandising
RETRIEVE PROJ-Q.	Preparations 500 Martinez 18000 Attractions

Hybrid Formatting

Hybrid formatting combines structured and tabular formatting. To retrieve data in hybrid form, embed SELECT statements as in structured output. However, use extended attributes for those items to be displayed in a table.

The next example uses the following declaration:

```
QD   PROJ-Q.
01   PROJ-REC.
     02   PF1       PIC X(20).
     02   PF2       PIC 9(11)   BINARY.
QD   PROJEMP-Q.
01   PROJEMP-REC.
     02   PF3       PIC X(20).
     02   PF4                   REAL.
     02   PF5       PIC X(20).
```

For example, assume you want to preserve the structure between PROJECT and PROJECT-EMPLOYEE as described under “Structured Formatting” in this section. However, you would like to process attributes of PROJECT-TEAM into a table. You would enter the following:

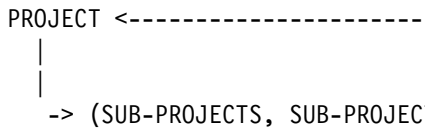
```
SELECT PROJ-Q FROM PROJECT
       (PF1 = PROJECT-TITLE, PF2 = PROJECT-NO
        SELECT PROJEMP-Q FROM PROJECT-TEAM
          WITH EMPLOYEE-SALARY > 17000
           (PF3 = LAST-NAME OF NAME, PF4 = EMPLOYEE-SALARY,
           PF5 = DEPT-TITLE OF DEPT-IN))
WHERE PROJECT-NO = 500.
```

Using the RETRIEVE statement on the left produces the data on the right:

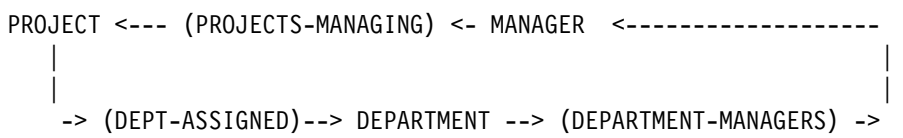
Statement	Data		
RETRIEVE PROJ-Q.	Preparations 500		
RETRIEVE PROJEMP-Q.	Jones	19000	Food Services
RETRIEVE PROJEMP-Q.	Smith	22000	Merchandising
RETRIEVE PROJEMP-Q.	Martinez	18000	Attractions

Transitive Closure

Transitive closure enables you to access a reflexive attribute. A reflexive attribute is an entity-valued attribute that refers to another entity in the same class. For example, the attribute PROJECT refers to an entity in the PROJECT class using the SUB-PROJECTS and SUB-PROJECT-OF attributes.



This reference can span many classes before it cycles back to itself. For example, the PROJECT class has an attribute DEPT-ASSIGNED that refers to DEPARTMENT. DEPARTMENT has an attribute DEPARTMENT-MANAGERS that refers to MANAGER. MANAGER has an attribute PROJECTS-MANAGING that refers back to PROJECT.



You can recursively access a reflexive attribute or cycle back to a class through transitive closure. Each recursive access represents another level of the query tree.

An attribute can have many values at each level as the query is traversed. To control the values chosen by the query, use the following functions:

- TRANSITIVE. By itself, this function prepares a query variable for use with the SET LEVEL statement. It is also used with the END LEVEL function as described in the following text. The following example uses this function on the SUB-PROJECTS attribute of the PROJECT class:

  ```
  TRANSITIVE(SUB-PROJECTS)
  ```

- END LEVEL. This function controls the level at which a query ends its search. If you try to search below the designated end level, you receive a message stating that the query has reached the last level. Use this function with TRANSITIVE, as in the following example:

  ```
  TRANSITIVE(SUB-PROJECTS END LEVEL 3)
  ```

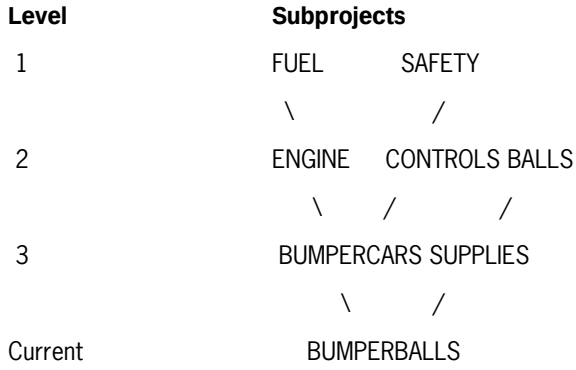
- SET LEVEL. This function controls the level at which the query searches. *SET LEVEL UP* moves the search up (away from the base of the tree) by one level from the level the query is on. *SET LEVEL DOWN* moves the search down (toward the base of the tree). The following example sets the level of the PROJ-Q query down:

  ```
  SET PROJ-Q LEVEL DOWN
  ```

These functions are further explained in the next example, which uses the following declaration:

```
QD   SUB-Q.
01   SUB-REC.
     02  SUBS  PIC X(20).
QD   PROJ-Q OF PROJECT.
```

Assume that the following structure illustrates the subprojects of the project BUMPERBALLS:



Each number represents a level in the tree. Each name is a subproject of the one below it. Assume you are looking for the subprojects of BUMPERBALLS. You first choose BUMPERBALLS with the following fragment:

```
SELECT PROJ-Q
  (SELECT SUB-Q FROM TRANSITIVE (SUB-PROJECTS)
    (SUBS = PROJECT-TITLE))
WHERE PROJECT-TITLE = "BUMPERBALLS".
RETRIEVE PROJ-Q.
```

CURRENT is explained under “CURRENT Function” in Section 5, “Designating Transaction State”

The RETRIEVE statement retrieves BUMPERBALLS, which represents the current level of the query. To choose the subproject of BUMPERBALLS, use the RETRIEVE statement on the SUB-Q query variable, as follows:

```
RETRIEVE SUB-Q.
```

This statement retrieves BUMPERCARS, the subproject of BUMPERBALLS. Repeating this statement retrieves SUPPLIES, another subproject of BUMPERBALLS.

To find the subprojects of SUPPLIES, set the level up from the query’s current level. You can use the following fragment after the previous fragment:

```
SET SUB-Q LEVEL UP BY 1.
RETRIEVE SUB-Q.
```

The RETRIEVE statement retrieves ENGINE. To return to the level of SUPPLIES once more, set the level down from the query's current level (level 2) as in the following fragment:

```
SET SUB-Q LEVEL DOWN BY 1
RETRIEVE SUB-Q.
```

The RETRIEVE statement retrieves SUPPLIES.

If you want the query to end its search at level 1, use the following statement instead of the first-mentioned SELECT statement:

```
SELECT PROJ-Q
  (SELECT SUB-Q FROM TRANSITIVE (SUB-PROJECTS END LEVEL 1)
    (SUBS = PROJECT-TITLE))
WHERE PROJECT-TITLE = "BUMPERBALLS".
RETRIEVE PROJ-Q.
```

If you try to search below level 1 (for example, by setting the level up to level 2), you receive a message stating that the query has reached the last level.

Section 5
Designating Transaction State

A database can be updated only through transactions. Transactions are statements bounded by begin transaction and end transaction statements. If a program part is so bounded, that part is in transaction state.

Within transaction state, SIM does not commit updates to the database until it reaches the end transaction statement. When an update is committed, it is applied permanently to the database and made visible to other users of the database.

SIM locks the entity used within a transaction so that simultaneous updates do not affect each other. The lock prevents other users from updating or retrieving an entity as it is being updated by the current user. Thus, to minimize the time a program is in transaction state, put only the update statements within transaction state. Do not use the begin transaction statement at the beginning of a program and the end transaction statement at the end of a program, because doing so can lock out other users until your program finishes.

This section discusses how to put a program in transaction state and how to designate and roll back to transaction points. Transaction points can cancel or save the results of statements within transaction state. This section also discusses entity-reference variables and the CURRENT function. Both can be used only in transaction state.

For more information on transactions, see volume 2 of the appropriate language reference manuals.

Starting Transaction State

The begin transaction statement begins transaction state as in the following example:

```
BEGIN-TRANSACTION
```

If you are performing a long transaction, you can append the EXCLUSIVE clause as follows:

```
BEGIN-TRANSACTION EXCLUSIVE
```

This clause ensures that you have exclusive access to the database and exclusive control of all structures in the database. However, before the program begins exclusive access, it must wait until all other programs that access the database have finished processing. Note that the EXCLUSIVE clause has a high performance cost and should be used sparingly.

You can then enter your update statements. Transaction state has the following restrictions:

- If you use a SELECT statement on a query variable within a certain transaction, you cannot use a RETRIEVE statement on that query variable outside the transaction. SIM closes the query at the end transaction statement, preventing you from using the selected entities.
- Entity-reference variables are valid only within the transaction in which they were retrieved.
- If you close a database involved in a transaction before the transaction ends, SIM ends the transaction without updating the database.
- You cannot update both a DMSII database and a SIM database in the same transaction. However, you can update both databases in the same program in different transaction states.

Designating Transaction Points

Transaction points assign markers at points in a transaction. You can use these transaction points to roll back an update within a transaction. To create a transaction point, use the save point statement and assign the point a number. For example, to create point 4, use the following statement:

```
SAVE TRANSACTION POINT 4
```

You now have a marker to which you can cancel any updates and from which processing continues. To cancel updates, use the cancel point statement. For example, assume you have updates marked as transaction points 4, 5, and 6. Assume that when you encounter a particular entity at point 6, you want to cancel all updates from transaction points 4 through 6. Use the following statement to cancel updates from the current marker back to point 4:

```
CANCEL TRANSACTION POINT 4
```

SIM undoes any changes made between the current point in the transaction and the point designated by the arithmetic expression.

Ending Transaction State

SIM does not commit any updates until the program leaves transaction state with an end transaction statement as follows:

```
END-TRANSACTION
```

At the end transaction statement, SIM also deactivates queries made active within transaction state.

If you prefer to cancel all the updates made in transaction state and then leave transaction state, use an abort transaction statement as follows:

```
ABORT-TRANSACTION
```

This statement also deactivates any queries made active within transaction state.

Using Entity-Reference Variables

In transaction state, you can refer explicitly to one entity through entity-reference variables and the CURRENT function. You can then compare entities directly with other entities instead of comparing entity values.

To refer to a specific entity, you must identify enough of that entity's attributes to choose the entity. In many cases, you need only identify a unique value for an entity's attribute to select the entity. For example, to choose an entity called Jones, you need only identify the name, as in the following selection expression:

```
WHERE LAST-NAME OF NAME = "JONES"
```

If JONES is not a unique name, you might also have to identify other attributes, such as EMPLOYEE-ID or EMPLOYEE-SALARY.

Entity-Reference Variables

An entity-reference variable refers explicitly to one entity and can be used only in transaction state. Entity-reference variables must be declared as described in “Declaring a Query” in Section 2, “Starting a Query Program.” A SELECT statement then chooses an entity for that variable, and a RETRIEVE statement assigns the entity to the variable. Whenever you need the entity, you can use the entity-reference variable instead.

For example, assume that a program declares an entity-reference variable called PROJ-ERV, as follows:

```
QD   PROJEMP-Q
01   PROJEMP-REC.
     02 PROJ-ERV USAGE ENTITYREFERENCE OF PROJECT.
                .
                .
                .
QD   ALLPROJ-Q.
01   ALLPROJ-REC.
     02 PROJECT-TITLE          PIC X(20).

WORKING-STORAGE SECTION.
01   E1-ERV ENTITYREFERENCE OF PROJECT.
```

Before using PROJ-ERV, you put the program in transaction state and choose a specific entity for it through a SELECT statement and a selection expression. The following example chooses the entity Smith for PROJ-ERV:

```
BEGIN-TRANSACTION.
  SELECT PROJEMP-Q FROM PROJECT-EMPLOYEE
      (PROJ-ERV = CURRENT-PROJECT)
       WHERE LAST-NAME OF NAME OF PROJECT-EMPLOYEE  = "SMITH".
```

The SELECT statement only prepares PROJ-ERV for the entity. To actually make PROJ-ERV contain the entity Smith, use a RETRIEVE statement with the query that uses the PROJ-ERV variable, as in the following example:

```
RETRIEVE PROJEMP-Q.
```

You can now use the variable in any statement throughout the program as a reference to the entity Smith. The reference does not change unless you retrieve PROJ-ERV again with the RETRIEVE statement.

The following example moves the entity Smith from the PROJ-ERV variable to the E1-ERV variable. A SELECT statement then chooses the names of all the projects worked on by the entity E1-ERV (Smith). The final statement ends the transaction state:

```
  MOVE PROJ-ERV TO E1-ERV.
  SELECT ALLPROJ-Q FROM PROJECT
       WHERE PROJECT FROM PROJECT = E1-ERV.
END-TRANSACTION.
```

To use a specific entity without declaring an entity-reference variable, use the CURRENT function, described in the following text.

Note: *You can use entity-reference variables only in the same transaction state in which they were assigned.*

CURRENT Function

The CURRENT function enables you to use the most recently selected entity of an active query. This function creates an unnamed entity-reference variable. You can use this function in retrieval and update queries, but only in the same transaction state in which you retrieved the query.

The next example uses the following declaration:

```
QD    MNGR-Q OF MANAGER.
QD    STAFF-Q.
01    STAFF-REC.
      02 E-LASTNAME PIC X(20).
```

For this example, assume that Smith is the entity that you want to select and retrieve, as follows:

```
BEGIN-TRANSACTION.
  SELECT MNGR-Q FROM MANAGER WHERE LAST-NAME OF NAME = "SMITH".
  RETRIEVE MNGR-Q.
```

Because Smith is the current entity of MNGR-Q, you can use Smith in any statement by referring to MNGR-Q with the CURRENT function. The following statement selects an entity from the EMPLOYEE class whose manager is Smith:

```
  SELECT STAFF-Q FROM EMPLOYEE
    (E-LASTNAME = LAST-NAME OF NAME)
    WHERE EMPLOYEE-MANAGER = CURRENT OF MNGR-Q
END-TRANSACTION.
```

Section 6
Updating Entities

Update queries can update entities only in transaction state. These queries can use the following statements:

- Update statements such as DELETE, INSERT, and MODIFY.
- Assignment statements such as INCLUDE, EXCLUDE, and ASSIGN.
- Multiple-statement update statements such as start and apply statements. Within these updates, queries can use any statement, including SELECT and RETRIEVE statements.

Update queries consist of a single statement or multiple statements and can contain statements that assign values to attributes. You can use update queries on immediate attributes only.

This section discusses the update statements, compares single- and multiple-statement update queries, defines the kinds of assignment statements, and describes error detection in queries.

Update Statements

Update statements are statements that insert, delete, and modify entities in a database. These statements include DELETE, INSERT, and MODIFY. They can be used with assignment statements, described under “Assignment Statements” later in this section.

DELETE Statement

This statement deletes one or more entities that satisfy the selection expression. A DELETE statement must have a query variable and a global selection expression. For more information on parts of a statement, see “Writing a Query Statement” in Section 2, “Starting a Query Program.”

By default, a DELETE statement deletes only one entity. You can increase the number of entities to be deleted by stating a limit. However, if the number of entities that meet the selection expression exceeds the limit number, the entities are not deleted and you receive an error message. To delete an unlimited number of entities, use NOLIMIT.

The following example deletes up to 50 entities in the MANAGER class where the bonus is less than 100:

```
DELETE (LIMIT = 50) MANAGER WHERE BONUS < 100
```

.Deleting an entity from a class that is a superclass automatically deletes the entity from all its subclasses. For example, the superclass EMPLOYEE has the subclasses MANAGER and PROJ-EMP. Assume that you use the following statement:

```
DELETE EMPLOYEE WHERE LAST-NAME OF NAME = "SMITH".
```

If Smith is in the PROJECT-EMPLOYEE and MANAGER subclasses, this statement deletes him from three classes: EMPLOYEE, PROJECT-EMPLOYEE, and MANAGER. However, deleting an entity from a subclass does not affect its superclass. For example, the following statement deletes Smith from the PROJECT-EMPLOYEE class but not from the EMPLOYEE class:

```
DELETE PROJECT-EMPLOYEE WHERE LAST-NAME OF NAME = "SMITH".
```

INSERT Statement

This statement inserts new entities and their immediate attributes into the database. It has two forms:

- INSERT. This form requires the name of the class into which the entity is inserted and one or more assignment statements that contain the attribute values of the entity. It inserts an entity that exists in no other class. The following example inserts an entity into the PROJECT class containing the values *225* for PROJECT-NO and *JANITORIAL* for PROJECT-TITLE:

```
INSERT PROJECT
  ASSIGN "JANITORIAL" TO PROJECT-TITLE
  ASSIGN 225 TO PROJECT-NO.
```

- INSERT FROM. This form assumes a superclass in which the entity plays a role. It inserts the entity from a superclass into a subclass. Thus, this form requires the name of the superclass after FROM and the name of the subclass before FROM. It also requires a global selection expression that selects only one entity. This form optionally requires assignment statements that contain new attribute values of the entity.

  The following example inserts an entity from the EMPLOYEE class into the MANAGER class. It assigns a value of 90 to BONUS.

```
INSERT MANAGER FROM EMPLOYEE
  WHERE LAST-NAME OF NAME = "SMITH"
  ASSIGN 90 TO BONUS
```

.Inserts affect only the class or superclass in the statement. They create all required superclass roles. They do not affect a subclass.

For more information on parts of a statement, see “Writing a Query Statement” in Section 2, “Starting a Query Program.”

MODIFY Statement

This statement modifies the attribute values of existing entities in the database. It requires the name of the class to which the modified entity belongs, assignment statements, and a global selection expression if the modified attribute is not a class attribute. The MODIFY statement can include a limit on the number of entries to be modified. For more information on parts of a statement, see "Writing a Query Statement" in Section 2, "Starting a Query Program."

The following example gives up to five employees a raise of 10 percent by multiplying their salaries by 1.1. If the number of salaries exceeds the limit, no entities are modified and you receive an error message.

```
MODIFY (LIMIT = 5) EMPLOYEE
      ASSIGN EMPLOYEE-SALARY * 1.1 TO EMPLOYEE-SALARY.
```

To modify an unlimited number of entities, use NOLIMIT.

Class Attributes

You can modify a class attribute, but you cannot insert or delete it. You cannot modify a class attribute if the same statement contains a standard attribute. A MODIFY command that updates a class attribute cannot have a WHERE clause. The following example modifies the class attribute NEXT-PROJECT-NO:

```
MODIFY PROJECT
    ASSIGN 87 TO NEXT-PROJECT-NO.
```

Entity-Valued Attributes

To modify an entity-valued attribute, an entity-valued expression must designate the value used to modify the attribute. This expression must evaluate one or more entities in the class that contains the entity-valued attribute.

The following example modifies the entity-valued attribute CURRENT-PROJECT in the PROJECT-EMPLOYEE class. It assigns the PROJECT of the entity Smith to the CURRENT-PROJECT of the entity Jones.

```
BEGIN-TRANSACTION.
   MODIFY PROJECT-EMPLOYEE
     ASSIGN PROJECT WITH LAST-NAME OF NAME
            OF PROJECT-EMPLOYEE = "SMITH"
       TO CURRENT-PROJECT
       WHERE LAST-NAME OF NAME OF PROJECT-EMPLOYEE = "JONES".
END-TRANSACTION.
```

Begin transaction and end transaction statements are described in Section 5, "Designating Transaction State."

Update Query Types

There are two types of update queries. The single-statement update consists of one update statement. The multiple-statement update consists of many update statements.

Single-Statement Update

A single-statement update is a single statement similar to that provided to an interactive user. SIM executes these statements when it encounters them and also activates and deactivates the query within the statement. Unlike error reporting for multiple-statement updates, in a single-statement update, SIM reports the errors immediately after a statement. Thus, single-statement updates are preferable to multiple-statement updates in terms of performance.

Almost all the update statements in this guide are single-statement updates. The following are examples of simple DELETE, INSERT, and MODIFY statements:

```
DELETE EMPLOYEE WHERE EMPLOYEE-SALARY = 15000.

INSERT MANAGER FROM EMPLOYEE WHERE LAST-NAME OF NAME = "SMITH".

MODIFY (LIMIT = 5) ASSIGNMENT
      ASSIGN "1/01/88" TO END-DATE WHERE START-DATE = "01/01/87".
```

Multiple-Statement Update

A multiple-statement update enables you to mix attribute assignments among other program statements, or to place assignments in other procedures and functions. You can use the multiple-statement update with the MODIFY and INSERT statements.

You use multiple-statement updates primarily on multivalued attributes, although you can use them on single-valued attributes. Because you do not know the number of values that need to be updated, you can put a conditional loop within a multiple-statement update and update as long as the condition is satisfied.

Multiple-statement updates begin with a start statement. This statement describes the nature of the selection, if any, and associates a query name with the query statement. The statement also activates the multiple-statement update query, enabling it to be processed by the system.

The following declaration is for the next examples:

```
QD   PROJEMP-Q OF PROJECT-EMPLOYEE.
```

The following example begins a multiple-statement MODIFY update on the PROJEMP-Q query for all entities in the PROJECT-EMPLOYEE class whose EMPLOYEE-SALARY equals $15,000:

```
START MODIFY PROJEMP-Q WHERE EMPLOYEE-SALARY = 15000.
```

You then use update statements anywhere within the program logic as long as they are executed after the start statement. The following example modifies entities in the PROJECT-EMPLOYEE class, which is associated with the PROJEMP-Q query. (The example assumes a LIMIT of 1, by default.) Assume DEPARTMENT-CHANGE and MY-DEPT-NAME are string variables. SIM does not apply these modifications until it processes an apply statement.

```
IF DEPARTMENT-CHANGE = "YES" THEN
   ASSIGN DEPARTMENT WHERE DEPT-TITLE = MY-DEPT-NAME
     TO DEPT-IN OF PROJEMP-Q.
```

To apply the multiple-statement update to the database, use an apply statement, which processes and deactivates the query. A DISCARD statement can also deactivate the query.

Note that SIM does not report run-time errors until the APPLY INSERT or APPLY MODIFY statement is executed. To detect the nature of these errors, call the DMEXCEPTIONINFO statement after the apply statement. For more information on the DMEXCEPTIONINFO statement, see “Detecting Errors in Queries” later in this section.

The following example applies all multiple-statement MODIFY updates in the PROJEMP-Q query that occurred after a start statement and before an apply statement:

```
APPLY MODIFY PROJEMP-Q.
```

Assignment Statements

Assignment statements add values to or remove values from attributes. Within a simple INSERT or MODIFY statement, they appear as clauses of INSERT or MODIFY. Within a multiple-statement INSERT or MODIFY update, they appear as statements. There are three assignment statements: ASSIGN, INCLUDE, and EXCLUDE.

ASSIGN Statement

This statement updates a value in a single-valued attribute. Used with an INSERT statement, ASSIGN assigns a new value to the attribute. Used with a MODIFY statement, ASSIGN replaces the previous value with a new value. The following example creates an entity in the PROJECT-EMPLOYEE class and assigns *SMITH* to its LAST-NAME attribute and *19000* to its EMPLOYEE-SALARY attribute:

```
INSERT PROJECT-EMPLOYEE
   ASSIGN ["SMITH" TO LAST-NAME] TO NAME
   ASSIGN 19000 TO EMPLOYEE-SALARY.
```

ASSIGN Statement with Compound Attributes

If you assign a value for a single-valued compound attribute, you must state all the parts of the compound. Otherwise, any omitted parts are not changed. In the following example, assume STREET, CITY, STATE, and ZIP are part of the compound attribute CURRENT-RESIDENCE in the PERSON class:

```
ASSIGN  ["21 OAK STREET" TO STREET, "COSTA MESA" TO CITY,
     "CA" TO STATE] TO CURRENT-RESIDENCE OF PERSON.
```

The example modifies STREET, CITY, and STATE only, omitting the ZIP part of the compound attribute EDUCATION. Thus, while STREET is assigned the value *21 OAK STREET*, CITY is assigned the value *COSTA MESA*, and state is assigned the value *CA*, ZIP is not assigned a new value.

INCLUDE Statement

This statement adds values to multivalued attributes by taking values from another multivalued attribute. Note that for entity-valued attributes, the class name of the attribute must follow the word INCLUDE.

The following example modifies the ASSIGNMENT-HISTORY attribute of the PROJECT class if the PROJECT-TITLE of PROJECT is FUTURE. It inserts the values of all entities with a START-DATE of 01/09/89 from the ASSIGNMENT class into the ASSIGNMENT-HISTORY attribute.

```
MODIFY PROJECT
       INCLUDE ASSIGNMENT WITH START-DATE "01/09/89"
       INTO ASSIGNMENT-HISTORY
WHERE PROJECT-TITLE = "FUTURE".
```

If including a value increases the number of values beyond the maximum limit defined for an attribute or invalidates any other defined option, the value is not added. The entire MODIFY statement is not processed.

INCLUDE Statement with Compound Attributes

If you include a value for a multivalued compound attribute, you must state all the parts of the compound. Otherwise, any omitted parts are assigned null values. In the following example, assume DEGREE-OBTAINED, YEAR-OBTAINED, and GPA are part of the compound attribute EDUCATION in the PERSON class:

```
INCLUDE  ["MS" TO DEGREE-OBTAINED, 3.5 TO GPA]
     TO EDUCATION OF PERSON.
```

The example modifies DEGREE-OBTAINED and GPA only, omitting the YEAR-OBTAINED part of the compound attribute EDUCATION. Thus, while DEGREE-OBTAINED is assigned the value MS and GPA is assigned the value 3.5, YEAR-OBTAINED is assigned a null value.

EXCLUDE Statement

This statement removes values from either single-valued or multivalued attributes. The LIMIT clause states the number of values to be removed. You can use the EXCLUDE statement with the MODIFY statement.

Note that when excluding entity-valued attributes, the attribute name must follow the word EXCLUDE.

The following example modifies the PROJECT class if the PROJECT-NO equals 500. It removes up to five assignments with an end date before 01/02/86 from the ASSIGNMENT-HISTORY attribute. If the number of assignments exceeds the limit, you receive an error and the query does not change anything.

```
MODIFY PROJECT
       EXCLUDE (LIMIT = 5) ASSIGNMENT-HISTORY WITH
       END-DATE < 01/02/86
       FROM ASSIGNMENT-HISTORY
WHERE PROJECT-NO = 500.
```

Detecting Errors in Queries

You can detect errors using normal program logic. Alternatively, you can detect errors in either retrieval or update queries by using the following SIM features:

- The exception word. Through this word, you can determine the type of error. Error categories are described in Appendix C, “Exception Fields and Categories.”
- DMEXCEPTIONMSG. This statement moves error message text into a variable.
- DMNEXTEXCEPTION. This statement causes the next error in the multiple error list to become the current error.
- DMEXCEPTIONINFO. This statement updates a structure in the program with additional information about the exception. For information about the record fields of this statement, see “DMEXCEPTIONINFO Record Field Names” in Appendix C, “Exception Fields and Categories.”

  You can use these statements after both retrieval and update query statements.

  The following example illustrates an error-handling statement:

```
RETRIEVE PROJEMP-Q.
IF DMSTATE THEN
   CALL SYSTEM DMEXCEPTIONMSG GIVING OUT-TEXT
   DISPLAY OUT-TEXT.
```

  The example retrieves PROJEMP-Q. If the exception word DMSTATE indicates an exception, the program calls DMEXCEPTIONMSG and moves the error message into the variable OUT-TEXT. It then displays the message by displaying the contents of OUT-TEXT.

  For more information on exception handling, refer to volume 2 of the appropriate language manual.

Appendix A
Sample Programs

This appendix shows sample COBOL74, ALGOL, and Pascal programs that use SIM program queries. The programs illustrate query concepts such as multiple-statement updates, hybrid formatting, and transitive closure.

The examples use the ORGANIZATION database, which is described in the *InfoExec SIM Technical Overview*.

Updating Project-Employee Projects

In each sample program, you can state what to do about each project-employee's projects in your database: add a project or drop a project. If you choose to drop a project, the program finishes up all related assignments by asking for ratings. The program then updates the overall rating of the project employees.

The samples show how to use multiple-statement updates, tabular retrieval, average function, single-statement updates, and the current function.

COBOL74 Version

```
ID DIVISION.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SPECIAL-NAMES.  RESERVE SEMANTIC.
DATA DIVISION.
DATA-BASE SECTION.
DB  PROJEMPDB = "ORGANIZATION" VALUE OF DBKIND IS SEMANTIC.
QD  PEMP-Q  OF PROJECT-EMPLOYEE.
QD  ASSI-Q.
01  ASSI-Q-REC.
    02   ASSI-START-DATE      PIC X(8).

WORKING-STORAGE SECTION.
01  PROJECT-INDICATOR         PIC X(4).
    88     PROJ-ADD                      VALUE "ADD".
    88     PROJ-DROP                     VALUE "DROP".
01  PROJ-NUM                  PIC 9(11)    BINARY.
01  SS-NUM                    PIC 9(11)    BINARY.
01  INPUT-RATING              PIC 9V9.
```

```
PROCEDURE DIVISION.
BEGIN.
    OPEN UPDATE PROJEMPDB.

** Input the PROJ-INDICATOR, PROJ-NUM, and SS-NUM.  **

    BEGIN-TRANSACTION.
    START MODIFY PEMP-Q WHERE PERSON-ID EQUAL SS-NUM.
    IF PROJ-ADD THEN
        INCLUDE PROJECT WITH PROJECT-NO EQUAL PROJ-NUM
          INTO CURRENT-PROJECT OF PEMP-Q
    ELSE
        IF PROJ-DROP THEN
          SELECT ASSI-Q FROM ASSIGNMENT
              ( ASSI-START-DATE = START-DATE )
            WHERE
              PERSON-ID OF STAFF-ASSIGNED
                EQUAL SS-NUM AND PROJECT-NO OF PROJECT-OF
                EQUAL PROJ-NUM AND NOT RATING EXISTS
          PERFORM PROCESS-AN-ASSIGNMENT THRU
                  PROCESS-AN-ASSIGNMENT-EXIT UNTIL DMSTATE
          ASSIGN DMFUNCTION AVG (RATING OF ASSIGNMENT-RECORD )
            TO OVERALL-RATING OF PEMP-Q
          EXCLUDE CURRENT-PROJECT WITH PROJECT-NO EQUAL PROJ-NUM
            FROM CURRENT-PROJECT OF PEMP-Q.
    APPLY MODIFY PEMP-Q.
    END-TRANSACTION.
    CLOSE PROJEMPDB.
    STOP RUN.

PROCESS-AN-ASSIGNMENT.
    RETRIEVE ASSI-Q
      ON EXCEPTION GO TO PROCESS-AN-ASSIGNMENT-EXIT.

** Display the START-DATE and enter the INPUT-RATING.  **

    MODIFY ASSIGNMENT
      ASSIGN INPUT-RATING TO RATING
        WHERE ASSIGNMENT EQUAL CURRENT OF ASSI-Q.

PROCESS-AN-ASSIGNMENT-EXIT.
    EXIT.
```

ALGOL Version

```
BEGIN
SEMANTIC DATABASE ORGANIZATION:
         (EMPLOYEE,MANAGER,PROJECT_EMPLOYEE,INTERIM_MANAGER,
          PROJECT,DEPARTMENT,ASSIGNMENT,PERSON);

FILE RMT(KIND = REMOTE,
         MAXRECSIZE = 360,
         FRAMESIZE = 8,
         MYUSE = IO);

TYPE UNPACKED DMRECORD ASS1_REC_TYPE
              (INTEGER ASS1_START_DATE);

ASS1_REC_TYPE ASS1_REC;

QUERY ASS1_Q(ASS1_REC),
      PEMP_Q(PROJECT_EMPLOYEE);

DEFINE        PROJ_ADD  = "ADD "#,
              PROJ_DROP = "DROP"#;

EBCDIC ARRAY PROJ_INDICATOR[0:3];

INTEGER PROJ_NUM,
        SS_NUM,
        MSG_LENGTH;

REAL    INPUT_RATING;
BOOLEAN QUERY_RESULT;

ARRAY MESSAGE_ARRAY[0:45],
      LANG_ARRAY[0:5];

DEFINE ABORT_GRACEFULLY =
       MYSELF.STATUS:= -1#;

PROCEDURE PROCESS_THE_MESSAGE;
%         -------------------
    BEGIN
    DMEXCEPTIONMSG(LANG_ARRAY, MESSAGE_ARRAY);
    MSG_LENGTH := MESSAGE_ARRAY[0];
    WRITE(RMT,MSG_LENGTH,POINTER(MESSAGE_ARRAY[1],8));
    END PROCESS_THE_MESSAGE;
```

```
PROCEDURE PROCESS_AN_ASSIGNMENT;
%         --------------------
    BEGIN
    QUERY_RESULT := RETRIEVE(ASS1_Q,ASS1_REC);

    WRITE(RMT,<I8,X4,F3.1>,ASS1_REC.ASS1_START_DATE,INPUT_RATING);
    MODIFY ASSIGNMENT
              (ASSIGN(RATING,INPUT_RATING))
      WHERE ASSIGNMENT = CURRENT(ASS1_Q);

    END PROCESS_AN_ASSIGNMENT;

% ---------------------------------------------------

    QUERY_RESULT := OPEN UPDATE ORGANIZATION;
    IF QUERY_RESULT THEN
       BEGIN
       PROCESS_THE_MESSAGE;
       ABORT_GRACEFULLY;
       END;

    BEGINTRANSACTION;

    READ(RMT,<A4,I6,I6,I6>,PROJ_INDICATOR,PROJ_NUM,SS_NUM,INPUT_RATING);

    STARTMODIFY (PEMP_Q WHERE PERSON_ID = SS_NUM);

    IF PROJ_INDICATOR = PROJ_ADD THEN
       INCLUDE(PEMP_Q.CURRENT_PROJECT,
                     [PROJECT WHERE PROJECT_NO = PROJ_NUM])
    ELSE
    IF PROJ_INDICATOR = PROJ_DROP THEN
       BEGIN
        SELECT ASS1_Q FROM ASSIGNMENT
               (ASS1_START_DATE = START_DATE)
               WHERE
                     STAFF_ASSIGNED.PERSON_ID = SS_NUM
                                  AND
                 PROJECT_OF.PROJECT_NO = PROJ_NUM
                                  AND
                 NOT DMEXISTS(RATING);

        WHILE NOT QUERY_RESULT DO
              PROCESS_AN_ASSIGNMENT;
```

```
IF QUERY_RESULT THEN
           IF REAL(QUERY_RESULT).DMEXCEPTION NEQ DMCOMPLETE THEN
               PROCESS_THE_MESSAGE;

         ASSIGN(PEMP_Q.OVERALL_RATING,DMAVG(ASSIGNMENT_RECORD.RATING));
         EXCLUDE(PEMP_Q.CURRENT_PROJECT,[CURRENT_PROJECT WHERE
                                         PROJECT_NO = PROJ_NUM]);
        END;
    APPLYMODIFY(PEMP_Q);
    ENDTRANSACTION;

    QUERY_RESULT:= CLOSE ORGANIZATION;

END.
```

Pascal Version

```
Program Updating_Employees ( Organization : database,
        MyDD : dictionary <functionname = 'DICTIONARYSYS372'>,
        Input, Output );

FROM Organization IMPORT
           Employee, Manager, Project_Employee, Interim-Manager,
           Project, Department, Assignment, Person;

TYPE
     Ass1_Type = RECORD
                  Ass1_Start_Date: integer;
                 END;

     Ass1_Rec_Type = DmRecord( Ass1_Type );


VAR
     Ass1_Rec :  Ass1_Rec_Type;

     Ass1_q : Query( Ass1_Rec_Type );
     Pemp_q : Query( Project_Employee );

     Proj_Indicator : packed array[1..4] of char;
     Proj_Num : integer;
     Ss_Num : integer;
     Input_Rating : real;

     Query_Result : DmStateType;
     Message_Array : string( 156 );
     Lang_Array : string( 17 );

CONST
     Proj_Add  = 'ADD ';
     Proj_Drop = 'DROP';

PROCEDURE Process_The_Message;
     BEGIN
     DmExceptionMsg( Lang_Array, Message_Array );
     WRITELN( Message_Array );
     END;  (* PROCESS_THE_MESSAGE *)
```

```
PROCEDURE Process_An_Assignment;
     BEGIN
     Query_Result := RETRIEVE( Ass1_q, Ass1_Rec );

     WRITELN( Ass1_Rec.Ass1_Start_Date, Input_Rating );
     MODIFY Assignment
            ( Rating := Input_Rating )
      WHERE Assignment = CURRENT(Ass1_q);

     END; (* Process_An_Assignment *)

(* ------------------------------------------------------------------ *)

     BEGIN
     Query_Result := OPEN( Organization, update );
     IF Boolean( Query_Result.DmErrorFlag ) THEN
        BEGIN
        Process_The_Message;
        ABORT;
        END;

     BeginTransaction;

     READLN( Proj_Indicator, Proj_Num, Ss_Num, Input_Rating );

     StartModify ( Pemp_q WHERE Person_Id = Ss_Num );

     IF Proj_Indicator = Proj_Add THEN
        INCLUDE( Pemp_q.Current_Project,
                       [Project WHERE Project_No = Proj_Num] )
     ELSE
     IF Proj_Indicator = Proj_Drop THEN
        BEGIN
        SELECT Ass1_q FROM Assignment
               ( Ass1_Start_Date = Start_Date )
                 WHERE ( Staff_Assigned.Person_Id = Ss_Num ) AND
                 ( Project_Of.Project_No = Proj_Num ) AND
                       ( NOT DmExists( Rating ) );

        WHILE NOT Boolean( Query_Result.DmErrorFlag ) DO
              Process_An_Assignment;

        IF Boolean( Query_Result.DmErrorFlag ) THEN
           IF integer( Query_Result.DmException ) <>
              DmExceptionRes( DmComplete ) THEN
               Process_The_Message;
```

```
            Pemp_q.Overall_Rating := DmAvg( Assignment_Record.Rating );
            EXCLUDE( Pemp_q.Current_Project,
                     [Current_Project
                       WHERE Project_No = Proj_Num] );
            END;
          ApplyModify( Pemp_q );
          EndTransaction;

          Query_Result := CLOSE( Organization );

END.
```

Archiving Assignments

Each sample program cleans up the database by removing assignments that were completed at least five years ago and by storing such assignments on tape.

The samples illustrate hybrid retrieval: the program formats some extended attributes in tabular form and others in structured form. The ALGOL and Pascal versions also pass query variables and query records as parameters.

COBOL74 Version

```
ID DIVISION.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SPECIAL-NAMES.  RESERVE SEMANTIC.
DATA DIVISION.
DATA-BASE SECTION.
DB  PROJEMPDB = "ORGANIZATION" VALUE OF DBKIND IS SEMANTIC.

QD  PEQ.
01  PEQ-REC.
    02  PEQ-NAME               PIC X(20).
    02  PERSON-ID              PIC 9(11)   BINARY.
    02  DEPT                   PIC X(20).
QD  AQ.
01  AQ-REC.
    02 AQ-START-DATE           PIC X(8).
    02  AQ-END-DATE            PIC X(8).
    02  AQ-EST-PERSON-HOURS    PIC 9(3).
    02  AQ-RATING              PIC 9(2)V9.
WORKING-STORAGE SECTION.
01  DEADLINE                   PIC X(8)    VALUE "01/01/88".
01  OUT-TEXT                   PIC X(156).

PROCEDURE DIVISION.
BEGIN.
    OPEN UPDATE PROJEMPDB.
    BEGIN-TRANSACTION.
    SELECT PEQ FROM PROJECT-EMPLOYEE

** PERSON-ID  does not need to be stated in the          **
** mapping list because the name of the field is the     **
** same as the name of the immediate attribute.          **

    ( PEQ-NAME = LAST-NAME OF NAME,
```

```
** DEPT is retrieved in tabular form with the rest of the **
** immediate attributes.                                   **

      DEPT = DEPT-TITLE OF DEPT-IN,
      SELECT AQ FROM ASSIGNMENT-RECORD

** The assignment attributes are retrieved as a           **
** substructure of PROJECT-EMPLOYEE.                      **

      ( AQ-START-DATE = START-DATE,
        AQ-END-DATE = END-DATE,
        AQ-EST-PERSON-HOURS = EST-PERSON-HOURS,
        AQ-RATING = RATING ) )
    WHERE SOME (END-DATE OF ASSIGNMENT-RECORD) BEFORE DEADLINE.
    PERFORM DO-EMPLOYEE THRU DO-EMPLOYEE-EXIT UNTIL DMSTATE.
    END-TRANSACTION.
    CLOSE PROJEMPDB.
    STOP-RUN.

DO-EMPLOYEE.
    RETRIEVE PEQ
      ON EXCEPTION
          GO TO DO-EMPLOYEE-EXIT.

** Write the employee information on tape.              **

    PERFORM DO-ASSIGNMENT THRU DO-ASSIGNMENT-EXIT UNTIL DMSTATE.

DO-EMPLOYEE-EXIT.
    EXIT.

DO-ASSIGNMENT.
    RETRIEVE AQ
      ON EXCEPTION
        IF DMEXCEPTION OF DMSTATE NOT = VALUE DMCOMPLETE THEN
           CALL SYSTEM DMEXCEPTIONMSG GIVING OUT-TEXT
           DISPLAY OUT-TEXT
           GO TO DO-ASSIGNMENT-EXIT
        ELSE
           GO TO DO-ASSIGNMENT-EXIT.

** Write the assignment information to tape.                **

    DELETE ASSIGNMENT WHERE ASSIGNMENT EQUAL CURRENT OF AQ.

DO-ASSIGNMENT-EXIT.
    EXIT.
```

ALGOL Version

```
BEGIN
SEMANTIC DATABASE ORGANIZATION:
         (EMPLOYEE,MANAGER,PROJECT_EMPLOYEE,INTERIM_MANAGER,
          PROJECT,DEPARTMENT,ASSIGNMENT,PERSON);

TYPE UNPACKED DMRECORD PEQ_REC_TYPE
              (EBCDIC ARRAY PEQ_NAME[0:19];
               INTEGER PERSON_ID;
               EBCDIC ARRAY DEPT[0:19]);

TYPE UNPACKED DMRECORD AQ_REC_TYPE
              (INTEGER AQ_START_DATE;
               INTEGER AQ_END_DATE;
               REAL    AQ_RATING);

PEQ_REC_TYPE  PEQ_REC;
AQ_REC_TYPE   AQ_REC;

QUERY PEQ(PEQ_REC),
      AQ(AQ_REC);

DEFINE  DEADLINE = 19880101#;% 01/01/1988

ARRAY OUT_TEXT[0:26],
      LANG_ARRAY[0:5];

BOOLEAN QUERY_RESULT;

INTEGER MSG_LENGTH;

FILE RMT(KIND = REMOTE,
         MAXRECSIZE = 360,
         FRAMESIZE = 8,
         MYUSE = IO);

DEFINE ABORT_GRACEFULLY =
       MYSELF.STATUS := -1#;

PROCEDURE PROCESS_THE_MESSAGE;
%         ------------------
    BEGIN
    DMEXCEPTIONMSG(LANG_ARRAY, OUT_TEXT);
    MSG_LENGTH := OUT_TEXT[0];
    WRITE(RMT,MSG_LENGTH,POINTER(OUT_TEXT[1],8));
    END PROCESS_THE_MESSAGE;
```

```
PROCEDURE DO_EMPLOYEE;
%         -----------
    BEGIN
    QUERY_RESULT := RETRIEVE(PEQ,PEQ_REC);

    IF NOT QUERY_RESULT THEN
       BEGIN
       WRITE(RMT,<A20,X4,I10,X4,A20>,
                   PEQ_REC.PEQ_NAME,PEQ_REC.PERSON_ID,PEQ_REC.DEPT);
       DO
         BEGIN
         QUERY_RESULT := RETRIEVE(AQ,AQ_REC);
         IF QUERY_RESULT THEN
            IF REAL (QUERY_RESULT).DMEXCEPTION NEQ DMCOMPLETE THEN
               PROCESS_THE_MESSAGE;
            ELSE
         ELSE
            BEGIN
            %  Archive the assignment to tape.
            DELETE ASSIGNMENT WHERE ASSIGNMENT = CURRENT(AQ);
            END;
         END
       UNTIL QUERY_RESULT;
       END
    ELSE
    IF REAL(QUERY_RESULT).DMEXCEPTION NEQ DMCOMPLETE THEN
       PROCESS_THE_MESSAGE;

    END DO_EMPLOYEE;


    QUERY_RESULT := OPEN UPDATE ORGANIZATION;
    IF QUERY_RESULT THEN
       BEGIN
       PROCESS_THE_MESSAGE;
       ABORT_GRACEFULLY;
       END;

    BEGINTRANSACTION;


    SELECT PEQ FROM PROJECT_EMPLOYEE
           (PEQ_NAME = NAME.LAST_NAME;
            % PERSON_ID need not be designated
            DEPT = DEPT_IN.DEPT_TITLE;
```

```
            SELECT AQ FROM ASSIGNMENT_RECORD
                   (AQ_START_DATE = START_DATE;
                    AQ_END_DATE = END_DATE;
                    AQ_RATING = RATING))
                  WHERE SOME (ASSIGNMENT_RECORD.END_DATE) < DEADLINE;

        DO
           DO_EMPLOYEE
        UNTIL QUERY_RESULT;

        ENDTRANSACTION;

        CLOSE ORGANIZATION;

    END.
```

Pascal Version

```
Program Archiving ( Organization : database,
        MyDD : dictionary <functionname = 'DICTIONARYSYS372'>,
        Output );

FROM Organization IMPORT
          Employee, Manager, Project_Employee, Interim_Manager,
          Project, Department, Assignment, Person;

TYPE
     Peq_Type = RECORD
                 Peq_Name : packed array[1..20] of char;
                 Person_Id : integer;
                 Dept: packed array[1..20] of char;
                END;

     Peq_Rec_Type = DmRecord( Pec_Type );

     Aq_Type = RECORD
                Aq_Start_Date : integer;
                Aq_End_Date : integer;
                Aq_Rating : real;
               END;

     Aq_Rec_Type = DmRecord( Aq_Type );

VAR
     Peq_Rec : Peq_Rec_Type;
     Aq_Rec : Aq_Rec_Type;

     Peq : Query( Peq_Rec_Type );
     Aq : Query( Aq_Rec_Type );

     Out_Text : string( 156 );
     Lang_Array : string( 17 );
     Query_Result : DmStateType;

CONST
     Deadline = '01/01/1988';

PROCEDURE Process_The_Message;
     BEGIN
     DmExceptionMsg( Lang_Array, Out_Text );
     WRITELN( Out_Text );
     END; (* Process_The_Message *)
```

```
PROCEDURE Do_Employee;
     BEGIN
     Query_Result := RETRIEVE( Peq, Peq_Rec );

     IF NOT Boolean( Query_Result.DmErrorFlag ) THEN
        BEGIN
        WRITELN( Peq_Rec.Peq_Name, Peq_Rec.Person_Id, Peq_Rec.Dept );
        WHILE NOT Boolean( Query_Result.DmErrorFlag ) DO
          BEGIN
          Query_Result := Retrieve( Aq, Aq_Rec );
          IF Boolean( Query_Result.DmErrorFlag ) THEN
             IF integer( Query_Result.DmException ) <>
                DmExceptionRes( DmComplete ) THEN
                Process_The_Message
             ELSE
          ELSE
             BEGIN
             (* Archive the assignment to tape then Delete *)
             DELETE Assignment WHERE Assignment = CURRENT( Aq );
             END;
          END
        END
     ELSE
     IF integer( Query_Result.DmException ) <>
                DmExceptionRes( DmComplete ) THEN
        Process_The_Message;
     END;  (* DO_EMPLOYEE *)

     BEGIN
     Query_Result := OPEN( Organization, Update );
     IF Boolean( Query_Result.DmErrorFlag ) THEN
        BEGIN
        Process_The_Message;
        ABORT;
        END;

     BeginTransaction;

     SELECT Peq FROM Project_Employee
            ( Peq_Name = Name.Last_Name;
                                 (* PERSON_ID need not be designated *)
              Dept = Dept_In.Dept_Title;
```

```
                SELECT Aq FROM Assignment_Record
                       (Aq_Start_Date = Start_Date;
                        Aq_End_Date = End_Date;
                        Aq_Rating = Rating ) )
                WHERE Some (Assignment_Record.End_Date) < Deadline;

        WHILE NOT Boolean( Query_Result.DmErrorFlag ) DO
           Do_Employee;

        EndTransaction;

        CLOSE( Organization );

END.
```

Listing Subprojects

Each sample program lists the subprojects for a specific project. This includes subprojects of subprojects.

These samples show how to use transitive closure and the related set statements.

COBOL74 Version

```
ID DIVISION.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SPECIAL-NAMES.  RESERVE SEMANTIC.
DATA DIVISION.
DATA-BASE SECTION.
DB  PROJEMPDB = "ORGANIZATION" VALUE OF DBKIND IS SEMANTIC.
QD  PQ.
01  PQ-REC.
    02  PQ-TITLE      PIC X(30).
QD  SQ.
01  SQ-REC.
    02  SQ-TITLE      PIC X(30).
WORKING-STORAGE SECTION.
01  MASTER-PROJECT    PIC X(30).
01  COUNTER           PIC 9(11)  BINARY VALUE 0.
01  DONE-STATUS       PIC X(3).
    88  NOT-DONE                      VALUE "NO".
    88  ALL-DONE                      VALUE "YES".

PROCEDURE DIVISION.
MAIN.
    OPEN PROJEMPDB.

** Enter the MASTER-PROJECT.                                    **

    SELECT PQ FROM PROJECT
      (PQ-TITLE = PROJECT-TITLE
       SELECT SQ FROM TRANSITIVE (SUB-PROJECTS)
         (SQ-TITLE = PROJECT-TITLE))
    WHERE PROJECT-TITLE OF PROJECT EQUAL MASTER-PROJECT.
    RETRIEVE PQ
      ON EXCEPTION
        DISPLAY "NO SUCH PROJECT".
      GO TO MAIN-END.
    MOVE "NO" TO DONE-STATUS.
    PERFORM GET-SUBPROJECT UNTIL ALL-DONE.

MAIN-END.
    CLOSE PROJEMPDB.
    STOP-RUN.
```

```
GET-SUBPROJECT.

RETRIEVE SQ.

** Display SQ-TITLE.                                              **

** Check the next level of subprojects.                           **

    SET SQ LEVEL UP.
    ADD 1 TO COUNTER.
    RETRIEVE SQ
      ON EXCEPTION
        IF DMEXCEPTION OF DMSTATE = VALUE DMCOMPLETE THEN
          PERFORM BACK-UP-LEVEL UNTIL ALL-DONE OR NOT DMSTATE
          GO TO GET-SUBPROJECT-EXIT
        ELSE
          MOVE "YES" TO DONE-STATUS
          GO TO GET-SUBPROJECT-EXIT.

GET-SUBPROJECT-EXIT.
EXIT.

BACK-UP-LEVEL.

** Check the previous level for more subprojects.                **

    SET SQ LEVEL DOWN.
    SUBTRACT 1 FROM COUNTER.
    IF COUNTER EQUAL 0 THEN

** All subprojects are exhausted.                                 **

      MOVE "YES" TO DONE-STATUS
    ELSE
      RETRIEVE SQ
        ON EXCEPTION
          IF DMEXCEPTION OF DMSTATE NOT = VALUE DMCOMPLETE
            THEN MOVE "YES" TO DONE-STATUS.
```

ALGOL Version

```
BEGIN
SEMANTIC DATABASE ORGANIZATION:
         (EMPLOYEE,MANAGER,PROJECT_EMPLOYEE,INTERIM_MANAGER,
          PROJECT,DEPARTMENT,ASSIGNMENT,PERSON);

TYPE UNPACKED DMRECORD PEQ_REC_TYPE
              (EBCDIC ARRAY PQ_TITLE[0:29]);

TYPE UNPACKED DMRECORD SQ_REC_TYPE
              (EBCDIC ARRAY SQ_TITLE[0:29]);

PEQ_REC_TYPE  PEQ_REC;
SQ_REC_TYPE   SQ_REC;

QUERY PEQ(PEQ_REC),
      SQ(SQ_REC);

BOOLEAN ALL_DONE, QUERY_RESULT;

INTEGER COUNTER, MSG_LENGTH;

EBCDIC ARRAY MASTER_PROJECT[0:29];

ARRAY OUT_TEXT[0:26],
      LANG_ARRAY[0:5];

DEFINE ABORT_GRACEFULLY =
       MYSELF.STATUS := -1#;

FILE RMT(KIND = REMOTE,
         MAXRECSIZE = 360,
         FRAMESIZE = 8,
         MYUSE = IO);

PROCEDURE PROCESS_THE_MESSAGE;
%         ------------------
    BEGIN
    DMEXCEPTIONMSG(LANG_ARRAY, OUT_TEXT);
    MSG_LENGTH := OUT_TEXT[0];
    WRITE(RMT,MSG_LENGTH,POINTER(OUT_TEXT[1],8));
    END PROCESS_THE_MESSAGE;
```

```
PROCEDURE BACK_UP_LEVEL;
          -------------
%
    BEGIN            % Check the previous level for more subprojects.
    SETTOPARENT(SQ);
    COUNTER:=-1;
    IF COUNTER LEQ 0 THEN
       ALL_DONE := TRUE
    ELSE
       QUERY_RESULT := RETRIEVE (SQ,SQ_REC);
    IF REAL(QUERY_RESULT).DMEXCEPTION=DMCOMPLETE THEN
       ALL_DONE := TRUE;
    END BACK_UP_LEVEL;

PROCEDURE GET_SUBPROJECT;
%         --------------
    BEGIN
    WRITE(RMT,<A30>,SQ_REC.SQ_TITLE);
    SETTOCHILD(SQ);
    COUNTER := * + 1;

    QUERY_RESULT := RETRIEVE(SQ,SQ_REC);

    IF QUERY_RESULT THEN
       IF REAL(QUERY_RESULT).DMEXCEPTION = DMCOMPLETE THEN
          DO
             BEGIN
             SETTOPARENT(SQ);
             COUNTER := *+1;
             IF COUNTER LEQ 0 THEN
                ALL_DONE := TRUE
             ELSE
                BEGIN
                QUERY_RESULT :=RETRIEVE(SQ,SQ_REC);
                IF REAL(QUERY_RESULT).DMEXCEPTION = DMCOMPLETE THEN
                   BACK_UP_LEVEL
                ELSE
                   ALL_DONE := TRUE;
                END
             END
          UNTIL ALL_DONE OR NOT QUERY_RESULT
       ELSE
          ALL_DONE := TRUE;

    END GET_SUBPROJECT;
```

```
    QUERY_RESULT := OPEN UPDATE ORGANIZATION;
    IF QUERY_RESULT THEN
       BEGIN
       PROCESS_THE_MESSAGE;
       ABORT_GRACEFULLY;
       END;

%**** Enter the master project *****

    SELECT PEQ FROM PROJECT
       (PQ_TITLE = PROJECT_TITLE;
        SELECT SQ FROM TRANSITIVE(SUB_PROJECTS)
          (SQ_TITLE = PROJECT_TITLE))
    WHERE PROJECT_TITLE = STRING8(MASTER_PROJECT[0],30);

    QUERY_RESULT := RETRIEVE(PEQ,PEQ_REC);
    IF QUERY_RESULT THEN
       BEGIN
       WRITE(RMT,<A15>,"No such project");
       ALL_DONE := TRUE;
       END
    ELSE
       BEGIN
       QUERY_RESULT := RETRIEVE(SQ,SQ_REC);
       IF QUERY_RESULT THEN
          BEGIN
          WRITE(RMT,<A14>,"No subprojects");
          ALL_DONE := TRUE;
          END;
       END;
    IF NOT ALL_DONE THEN
       DO
          GET_SUBPROJECT
       UNTIL ALL_DONE;

    CLOSE ORGANIZATION;

END.
```

Pascal Version

```
Program Listing_Subprojects ( Organization : database,
        MyDD : dictionary <functionname = 'DICTIONARYSYS372'>,
        Output );

FROM Organization IMPORT
          Employee, Manager, Project_Employee, Interim_Manager,
          Project, Department, Assignment, Person;

TYPE
     Peq_Type = RECORD
                 Pq_Title : packed array[1..30] of char;
                END;

     Peq_Rec_Type = DmRecord( Peq_Type );

     Sq_Type = RECORD
                Sq_Title : packed array[1..30] of char;
               END;

     Sq_Rec_Type = DmRecord( Sq_Type );

VAR
     Peq_Rec :  Peq_Rec_Type;
     Sq_Rec : Sq_Rec_Type;

     Peq : Query( Peq_Rec_Type );
     Sq : Query( Sq_Rec_Type );

     All_Done : Boolean;
     Query_Result : DmStateType;
     Counter: integer;

     Master_Project : packed array[1..30] of char;

     Out_Text : String( 156 );
     Lang_Array : String( 17 );

PROCEDURE Process_The_Message;
     BEGIN
     DmExceptionMsg( Lang_Array, Out_Text );
     WRITELN( Out_Text );
     END; (* Process_The_Message *)
```

```
PROCEDURE Back_Up_Level;
     BEGIN        (* CHECK THE PREVIOUS LEVEL FOR MORE SUBPROJECTS *)
     SetToParent( Sq );
     Counter:=-1;
     IF Counter <= 0 THEN
        All_Done:=TRUE
     ELSE
        Query_Result:=RETRIEVE( Sq, Sq_Rec );
     IF integer( Query_Result.DmException ) =
        DmExceptionRes( DmComplete ) THEN
        All_Done:=TRUE;
     END;  (* Back_Up_Level *)

PROCEDURE Get_Subproject;
     BEGIN
     WRITELN( Sq_Rec.Sq_Title );
     SetToChild( Sq );
     Counter:=Counter + 1;
     Query_Result := RETRIEVE( Sq, Sq_Rec );

     IF Boolean( Query_Result.DmErrorFlag ) THEN
        IF integer( Query_Result.DmException ) =
           DmExceptionRes( DmComplete ) THEN
           WHILE NOT All_Done OR Boolean( Query_Result.DmErrorFlag ) DO
              BEGIN
              SetToParent( Sq );
              Counter:=Counter + 1;
              IF Counter <= 0 THEN
                 All_Done := TRUE
              ELSE
                 BEGIN
                 Query_Result := RETRIEVE( Sq, Sq_Rec );
                 IF integer( Query_Result.DmException ) =
                     DmExceptionRes( DmComplete ) THEN
                     Back_Up_Level
                 ELSE
                     All_Done:=TRUE;
                 END
              END
        ELSE
          All_Done:=TRUE;
     END;  (*GET_SUBPROJECT *)

     BEGIN
     Query_Result := OPEN( Organization, update );
     IF Boolean( Query_Result.DmErrorFlag ) THEN
        BEGIN
        Process_The_Message;
        ABORT;
        END;
```

```
(****   Enter the master project ****)

     SELECT Peq FROM Project
         (Pq_Title = Project_Title;
          SELECT Sq FROM Transitive( Sub_Projects )
            (Sq_Title = Project_Title ))
     WHERE Project_Title = MASTER_PROJECT;

     Query_Result := RETRIEVE( Peq, Peq_Rec );
     IF Boolean( Query_Result.DmErrorFlag ) THEN
        BEGIN
        WRITELN('No such project');
        All_Done := TRUE;
        END
     ELSE
        BEGIN
        Query_Result := RETRIEVE( Sq, Sq_Rec );
        IF Boolean( Query_Result.DmErrorFlag ) THEN
           BEGIN
           WRITELN('No subprojects');
           All_Done := TRUE;
           END;
        END;
     IF NOT All_Done THEN
        WHILE NOT All_Done DO
           Get_Subproject;

     CLOSE( Organization );

END.
```

Appendix B
Language Comparison Tables

The following tables present the COBOL74, ALGOL, and Pascal constructs that correspond to the generic terms used in this guide. Table B–1 compares statements, and Table B–2 compares functions and operators. The generic terms are in alphabetical order. Note that only those terms that differ in each language are listed.

For more information on the language statements, consult the appropriate language manual.

Table B–1. Statement Comparison Table

Generic Term	COBOL74 Statement	ALGOL Statement	Pascal Statement
Abort transaction	ABORT-TRANSACTION	ABORTTRANSACTION	AbortTransaction
Apply	APPLY INSERT APPLY MODIFY	APPLYINSERT APPLYMODIFY	ApplyInsert ApplyModify
Begin transaction	BEGIN-TRANSACTION	BEGINTRANSACTION	BeginTransaction
Cancel point	CANCEL TRANSACTION POINT	CANCELTRPOINT	CancelTrPoint
Data description entries	Data description entries	DICTIONARY RECORD	Data description entries
DICTIONARY option	DICTIONARY IS clause	DICTIONARY option	Dictionary specification in header
End transaction	END-TRANSACTION	ENDTRANSACTION	EndTransaction
Entity qualification	Dictionary option	Entity qualification	Database attribute specification
Exception word	DMSTATE	Exception variable, Exception type	Predefined exception functions
Global selection expression	WHERE clause	WHERE clause	Where clause

Table B–1. Statement Comparison Table

Generic Term	COBOL74 Statement	ALGOL Statement	Pascal Statement
Local selection expression	WITH clause	WHERE clause, square brackets	Where clause, square brackets
Qualification	OF clause, parentheses	Qualification term	Dot notation
Query record description	Query record description	DMRECORD description	DmRecord description
Save point	SAVE TRANSACTION POINT	SAVETRPOINT	SaveTrPoint
Set	SET LEVEL DOWN SET LEVEL UP	SETTOPARENT SETTOCHILD	SetToParent SetToChild
Start	START INSERT START MODIFY	STARTINSERT STARTMODIFY	StartInsert StartModify

Table B–2. Function and Operator Comparison Table

Generic Term	COBOL74 Function	ALGOL Function	Pascal Function
Average	AVG	DMAVG	DmAvg
Character	CHR	DMCHR	Pascal hexadecimal string
Count	COUNT	DMCOUNT	DmCount
Equal to	EQUAL	EQL	=
	=	=	
Exists	EXISTS	DMEXISTS	DmExists
		EXISTS	Exists
Extract	EXT	DMEXT	DmExt
Greater than	GREATER	GTR	>
	>	>	
Greater than or equal to	(See note.)	GEQ	>=
		>=	
Is a	ISA	DMISA	DmIsa
Is in	ISIN	DMMATCH	DmMatch
Length	LEN	DMLENGTH	DmLength

Table B–2. Function and Operator Comparison Table

Generic Term	COBOL74 Function	ALGOL Function	Pascal Function
Less than	LESS	LSS	<
	<	<	
Less than or equal to	(See note.)	LEQ	<=
		<=	
Maximum	MAX	DMMAX	DmMax
Minimum	MIN	DMMIN	DmMin
Not equal	(See note.)	NEQ	<>
		^=	
Position	POS	DMPOS	DmPos
Predecessor	PRED	DMPRED	DmPred
Repeat	RPT	DMRPT	DmRpt
Successor	SUCC	DMSUCC	DmSucc
Sum	SUM	DMSUM	DmSum

Note: *The greater than or equal to, less than or equal to, and not equal constructs in COBOL74 are available only as a combination of two functions and/or operators.*

Appendix C
Exception Fields and Categories

This appendix describes the SIM DMSTATE exception word and its fields, the SIM DMSII result word, the DMEXCEPTIONINFO record and its fields, and the major exception categories and subcategories that are returned on an exception.

SIM DMSTATE Exception Word and Fields

For every SIM operation, exception information is stored in the SIM DMSTATE exception word. To access the information in this word from your program, you must use language extensions. In COBOL74 the information is stored automatically in the DMSTATE exception word, and you can access the exception information using the DMSTATE statement. In ALGOL and Pascal, you must assign a Boolean variable to contain the result so that you can access the exception information using the <exception field> syntax in ALGOL or the DmStateType record in Pascal.

Table C–1 describes the fields that the SIM DMSTATE exception word contains.

Table C–1. SIM DMSTATE Exception Word Fields

Field Name	Explanation
DMEXCEPTION	Contains a numeric value that identifies a major exception category. Refer to "Exception Categories" later in this appendix for more information on the types of major exception categories.
DMSUBEXCEPTION	Contains a numeric value that identifies a subcategory of a major exception category. Refer to "Exception Categories" later in this appendix for more information on the types of subcategories.
DMMOREEXCEPTION	Contains a Boolean value that indicates whether there are more errors for the SIM statement.
DMUPDATECOUNT	Contains the number of entities actually updated by an update query. If the query did not update any entities for reasons other than an error, DMWARNING subcategory 12 or 13 is returned.

SIM DMSII Result Word

When a physical database exception occurs, the DMSII result word is stored in a predefined word for each language. DMRESULT is used in COBOL74. DMSTATUS is used in ALGOL. DmExceptionInfoType is used in Pascal.

To determine the DMSII error category and subcategory, look in the result word at bit 35:8 for the category and bit 19:6 for the subcategory. You can access these bits as follows:

- In COBOL74, use the statements DMCATEGORY[35:7:8] and DMSUBCATEGORY[19:15:16].
- In ALGOL, use the statements DMEXCEPTIONRECORD.DMSTATUS.[35:8] and DMEXCEPTIONRECORD.DMSTATUS.[19:16].
- In Pascal, use the DmCategory and DmSubCategory fields.

For example, in a COBOL application, to put the values of these bits in individual data items, declare the following code in the WORKING-STORAGE SECTION:

```
01  WS-DMRESULT  USAGE REAL.
01  DMCATEGORY  USAGE REAL.
01  DMSUBCATEGORY  USAGE REAL.
```

To access the appropriate bits, use the following example:

```
MOVE DMRESULT TO WS-DMRESULT.
MOVE WS-DMSRESULT TO DMCATEGORY[35:7:8]
MOVE WS-DMRESULT TO DMSUBCATEGORY[19:15:16].
```

You can now determine the DMCATEGORY and DMSUBCATEGORY values.

For information on the DMSII error categories and subcategories, see the *DMSII Application Program Interfaces Programming Guide*.

DMEXCEPTIONINFO Record Field Names

The DMEXCEPTIONINFO record contains information about the exception that occurred that is different from the information in the DMSTATE exception word and the DMSII result word. To access the information in the DMEXCEPTIONINFO record from COBOL74 programs, use the CALL SYSTEM statement with the DMEXCEPTIONINFO parameter. In ALGOL, use the DMEXCEPTION record. In Pascal, use the DmExceptionInfoType record with the DmExceptionInfo function. Note that Pascal has additional fields that are not listed in Table C–2.

Table C–2 describes the fields that the DMEXCEPTIONINFO record contains.

Table C–2. DMEXCEPTIONINFO Record Field Names

Field Name	Explanation
DMSTRUCTURENAME	Contains the name of the DMSII structure on which a physical database exception was detected. It is meaningless for logical database exceptions.
DMLUCNAME	Contains the name of the SIM logical component on which a physical or logical database exception was detected.
DMDBNAME	Contains the internal database name on which a physical or logical database exception, a verification or constraint exception, or a transaction exception occurred.
DMVERIFYNAME	Contains the name of the VERIFY statement that caused the verification exception, or a description of the attribute option that caused the constraint exception.

Exception Categories

The remainder of this appendix lists and explains the category and subcategory information you might receive when processing SIM exceptions with the DMSTATE statement in COBOL74, the DMEXCEPTION and DMSUBEXCEPTION fields in ALGOL, and the DmStateType record in Pascal.

Category 0: DMNOERROR

There is only one subcategory in this category—0 (zero).

If you receive a category and a subcategory of 0 (zero), then an exception did not occur, and the operation you requested processed successfully. When you are calling DMNEXTEXCEPTION, this category shows that no further exceptions exist in the exception queue.

Category 1: DMWARNING

All exceptions with a category of 1 return warnings. A warning informs you about occurrences that do not affect the results of the operation that you processed. Table C–3 describes the subcategories associated with category 1.

Table C–3. Warning Messages

Subcategory	Explanation
0	There is no subcategory information.
1	The query was reparsed; however, the query was processed correctly. To obtain better performance, recompile your program.
2	Protocol translation is occurring. You can obtain better performance by recompiling your program.
3	An OML warning was issued. This message indicates that an error occurred in the SIM message handling.
4	The s-code compiler issued a warning while generating the code segment of your query.
10	The selection expression in the MODIFY query did not select any entities.
11	A local selection expression in an INCLUDE or EXCLUDE statement did not select any entities.
12	The query did not update any entities.
13	You are attempting to open a cursor that is currently active. This cursor will be closed and then reopened.
14	The item just returned has a null value.
15	The database has been opened for inquiry-purposes only.

Category 2: DMCOMPLETE

This category indicates that a sequence of events has completed. Table C–4 describes the subcategories associated with category 2.

Table C–4. Completion Messages

Subcategory	Explanation
0	The sequence of events you requested has completed and there is no additional information to be returned.
1	Your query is complete and there is no more data to be returned.
2	There is no more data to return at this level of the query structure. However, there might be more data at other levels.
3	The query processing was interrupted and can now be continued.

Category 3: DMFAILED

This category is returned when a query fails or when a process fails to complete. Table C–5 describes the subcategories associated with category 3.

Table C–5. Failure Messages

Subcategory	Explanation
0	The query or process failed to complete and there is no additional information to return.
1	The version timestamp of the current database has changed since you compiled your program.
2	Your program is incompatible with the current release of SIM. You need to recompile your program.
3	A syntax error exists in your query. Access the DMEXCEPTIONINFO record to obtain more information about the syntax error. Refer to "DMEXCEPTIONINFO Record Field Names" earlier in this appendix for more information.
4	The s-code compiler detected an error while generating the code segment for your query. This is an internal error. Please file a User Communication Form (UCF) to report this problem.
5	The query number you have designated is larger than the maximum query number allowed.
6	You have exceeded the limit on active user-defined functions.
7	You tried to enter a SIM environment that is currently being used by another process.
8	Your complex update query needs to be reparsed. Resubmit your query or recompile your program.
9	Your query cannot continue until the previously reported errors have been fixed.
10	The FROM clause of your INSERT statement selected more than one entity.
12	Your update query has been rejected because the number of entities selected exceeds the limit defined by the LIMIT clause.
13	Your query has been rejected because the number of entities selected exceeds the limit defined by the LIMIT clause of an EXCLUDE statement.
14	Your update query failed because it tried to assign a value that already exists in the database to an attribute declared with a UNIQUE constraint.
15	The named verify failed.
16	Your ASSIGN statement is not valid for a STARTINSERT query. (This exception can be returned only if you are using the host language extensions to query the database.)
17	Your query failed because you tried to add a subrole that already exists.

Table C–5. Failure Messages

Subcategory	Explanation
20	Your query failed because the database you want to use is not open.
21	Your request to open the database failed because the database is already open.
22	Your request to open a database failed because the database cannot be found.
23	Your request to open a database failed because too many databases are already open.
24	Your update query failed because the database is open for inquiry purposes only.
26	The DMSIISUPPORT or FILESUPPORT library required to access the database or file is not available. Please verify your configuration specifications for the SIMDMSIISUPPORT and the SIMFILESUPPORT library functions. Note, that this error should not occur. If it does, there might be an installation problem with the SLCONFIGSUPPORT library.
30	Your program must be in transaction state to accomplish the requested action.
31	Your transaction cannot start because another transaction is already in process.
32	You tried to update more than one database in a single update query. Updates within a single transaction must all occur against the same database.
33	The operation you requested failed because the database has been aborted.
34	The operation you requested failed because a deadlock occurred; either a MAXWAIT timeout or record lock caused the deadlock.
39	You designated an invalid savepoint number.
40	You have not initialized the query variable.
41	You requested a query variable that is in use.
42	The query variable for the CURRENT function is not open.
43	The query variable for the CURRENT function does not reference an entity of the correct class.
44	The query variable in a RETRIEVE statement refers to a query record description that is not valid at the current time.
49	You need to designate an ADDS dictionary.
50	You attempted to bind programs that do not use the same ADDS dictionary. Programs that you want to bind together must use the same data dictionary.
51	You have designated either an invalid ADDS dictionary name or a dictionary that is not defined.

Table C–5. Failure Messages

Subcategory	Explanation
52	You do not have the required privileges to query the ADDS dictionary.
53	You tried to use a database that is not compatible with SIM.
54	You have provided an invalid dictionary name.
55	You have designated a symbolic value that is either invalid or outside of the allowed range.
56	You tried to access a database that has not been generated.
57	The database you want to use is not defined in an ADDS dictionary.
58	Your query contains requests that are valid only for retrievals on LINC II databases.
59	The database you want to update can be opened for inquiry purposes only.
60	You tried to assign a value to a preexisting role. Attributes of preexisting roles cannot be assigned values.
61	The DMSIISUPPORT library is suspended. The text of this message includes the mix number you should investigate.
62	You tried to use a database that was generated using software from an earlier software release than the software release currently in use. Use the SIM Utility UPDATE command to regenerate your database.
63	You tried to use a database that was generated using software from a later software release than the software release currently in use. You can only use the database with the newer software release.
65	The type of data value you are supplying for the item is not correct. For example, you might be supplying an integer when a string is expected.
72	Your query designates an invalid class name.
74	An error occurred while calling a CENTRALSUPPORT function.
75	A program variable contains an incorrectly formed numeric value.
77	You tried to add a value to a multivalued attribute that was already defined for another occurrence of the multivalued attribute in the same entity.
78	You designated a value for an attribute that is outside the range allowed for that attribute. Check the schema definition to determine the valid range of values for the attribute.
79	The program could not get a message from the CENTRALSUPPORT library.
80	You tried to assign a new value to the STATIC option. An attribute with the STATIC option can be assigned a new value only when its current value is null.
86	You cannot save decomposed assignments in a saved query file. This message can be returned only if you are using the host language extensions to query the database.

Table C–5. Failure Messages

Subcategory	Explanation
92	The database cannot be opened because the keys for the SIM software are not present.
96	You are trying to represent a Boolean value with something other than 0 (zero) or 1. The value used to represent a Boolean value must evaluate to 0 or 1.
103	Your query was rejected because you tried to assign more than one entity to a single-valued entity-valued attribute (EVA).
104	You tried to use a required attribute for which no value was assigned.
105	The date value was invalid or did not follow the form MM/DD/YY or MM/DD/YYYY.
106	The time value was invalid or did not follow the form HH:MM:SS or HH:MM.
107	The date increment you requested produced an invalid date. That is, the date is either less than 00/00/000 or greater than 12/31/9999.
108	An entity was not selected because the Remap and Test (RAT) process failed.
109	The operation you requested failed because there is no current master entity.
110	The query failed because it uses an invalid symbolic string.
111	You tried to duplicate an attribute that requires no duplicates.
112	You tried to assign a value that already exists to a unique attribute.
113	You tried to assign a value to an attribute that must be distinct.
114	The user limit was exceeded.
115	The cursor limit was exceeded.
116	The logical underlying component (LUC) number of a SIM component is invalid.
117	The schema is incorrect.
118	You tried to use a feature that is not supported.
119	You tried to use a database title that is invalid.
120	You tried to use a title of a database that does not exist.
121	The database limit was exceeded.
122	The database control file has an error.
123	The compilation of the RAT process failed.
124	You tried to use a feature that is not compiled.
125	The security space was invalid.

Table C–5. Failure Messages

Subcategory	Explanation
126	Your query produced an incorrect NOTFOUND message while trying to access a class attribute.
127	The tank file has an input/output error.
128	The program produced an error while unlocking the database.
129	The program produced an error while locking the database.
130	A directory update has failed. The record to be written to the directory is incorrect.
131	The system cannot determine the value to allocate to the next surrogate.
132	The limit for the DMSIISUPPORT dynamic code has been reached.
133	You tried to use a database number that is invalid.
134	You tried to access a database that is not open.
135	An SCODE fault has occurred.
136	The entity is not locked.
137	The master record is invalid.
138	The cursor is invalid.
139	The parameter is invalid.
140	You tried to open a database that is already open.
141	The database can be accessed only for validating Object Manipulation Language (OML) syntax.
142	The DMSIISUPPORT software you are using does not match the rest of your SIM software.
143	You tried to use a class for which you have no access.
144	You tried to use an attribute for which you have no access.
145	You are trying to use software that is incompatible with your other software.
146	You tried to create a subrole that already exists for an entity.
147	You tried to use a superclass entity that is missing.
148	A transaction was not completed.
149	The program cannot continue.
150	A transaction failed and the only way to correct the database is to roll back the transaction.
151	You tried to assign a value that would have exceeded the maximum number of occurrences for an entity-valued attribute (EVA).
152	You tried to assign a value that would have exceeded the maximum number of occurrences for a data-valued attribute.

Table C–5. Failure Messages

Subcategory	Explanation
153	You tried to insert a subclass that would have exceeded the maximum number of occurrences for a multivalued subrole.
154	You tried to create an entity that was missing a required subrole. In this case, the subrole must be inserted from the perspective of the subclass.
155	Your update failed because a required multivalued data-valued attribute is missing.
156	Your update failed because a required multivalued entity-valued attribute (EVA) is missing.
157	Your update failed because a required single-valued entity-valued attribute (EVA) is missing.
158	A required source value is missing from your update.
159	You tried to use a dictionary that does not exist.
160	The program produced an error while trying to access a file description.
161	The build tables have an error.
162	Your update opened a LINC database rather than a SIM database.
163	The database has been validated but not generated.
164	The database can be opened for inquiry only.

Category 4: DMSYSTEM

This category indicates exceptions that are of a fatal nature to your program and possibly to SIM. If you receive these exceptions, you should either close the database or issue a DS (Discontinue) system command to stop your program. Table C–6 describes the subcategories associated with category 4.

Table C–6. Fatal Messages

Subcategory	Explanation
0	A fatal system error occurred. There is no additional information available with this subcategory.
1	A software protocol violation occurred. SIM encountered an internal fault while processing your request. Please file a UCF to report this problem.
2	You attempted to use a feature that is not currently implemented.
3	SIM encountered an internal fault while processing your request.
4	A system integrity failure occurred. This message is followed by another message that provides more detailed information.
5	An error occurred during the logical database processing.
6	An error occurred during the physical database processing.
7	An error occurred during the processing of a data dictionary request. This message provides an ADDS error code and subcode. Refer to the SYMBOL/ADDS/PROPERTIES file.
8	A work file failure occurred. This message provides an error code and subcode that is returned by the I/O subsystem. Refer to the *System Messages Support Reference Manual* for more information.
9	SIM encountered an internal fault while processing your request. Please file a UCF to report this problem.

Index

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W