1986-12

The implementation of a functional/daplex interface for the multi-lingual database system.

Lim, Beng Hock

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

THE IMPLEMENTATION OF A
FUNCTIONAL/DAPLEX INTERFACE FOR THE
MULTI-LINGUAL DATABASE SYSTEM

by

Beng Hock Lim

December 1986

Thesis Advisor: David K. Hsiao

unclassified
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a REPORT SECURITY CLASSIFICATION unclassified	1b RESTRICTIVE MARKINGS
2a SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.
2b DECLASSIFICATION/DOWNGRADING SCHEDULE	
4 PERFORMING ORGANIZATION REPORT NUMBER(S)	5 MONITORING ORGANIZATION REPORT NUMBER(S)
6a NAME OF PERFORMING ORGANIZATION Naval Postgraduate School 6b OFFICE SYMBOL (If applicable) 52	7a NAME OF MONITORING ORGANIZATION Naval Postgraduate School
6c ADDRESS (City, State, and ZIP Code) Monterey, California 93943-5000	7b ADDRESS (City, State, and ZIP Code) Monterey, California 93943-5000
8a NAME OF FUNDING/SPONSORING ORGANIZATION 8b OFFICE SYMBOL (If applicable)	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c ADDRESS (City, State, and ZIP Code)	10 SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO / PROJECT NO / TASK NO / WORK UNIT ACCESSION NO

11 TITLE (Include Security Classification)
THE IMPLEMENTATION OF A FUNCTIONAL/DAPLEX INTERFACE FOR THE MULTI-LINGUAL DATABASE SYSTEM

12 PERSONAL AUTHOR(S) Lim, Beng Hock

13a TYPE OF REPORT	13b TIME COVERED	14 DATE OF REPORT (Year, Month, Day)	15 PAGE COUNT
Master's Thesis	FROM _____ TO _____	1986 December	206

16 SUPPLEMENTARY NOTATION

17 COSATI CODES			18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	database system; multi-lingual database system (MLDS); functional/Daplex language interface

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

Traditionally, the design and implementation of a conventional database system begins with the choice of a data model followed by the specification of a model-based data language. Thus, the database system is restricted to a single data model and a specific data language. An alternative to this traditional approach to database-system development is the multi-lingual database system (MLDS). This alternative approach enables the user to access and manage a large collection of databases via several data models and their corresponding data languages without the aforementioned restriction.

In this thesis we present the implementation of a functional/Daplex

20 DISTRIBUTION/AVAILABILITY OF ABSTRACT	21 ABSTRACT SECURITY CLASSIFICATION	
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	unclassified	
22a NAME OF RESPONSIBLE INDIVIDUAL Prof. D.K. Hsiao	22b TELEPHONE (Include Area Code) (408) 646-2253	22c OFFICE SYMBOL Code 52Hq

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted
All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE
unclassified

18. attribute-based data language (ABDL)

19. language interface for MLDS. Specifically, we present the implementation of an interface which translates Daplex language calls into attribute-based data language (ABDL) requests which, as the kernel language, support all other data language interfaces.

Approved for public release: distribution is unlimited.

The Implementation of a Functional/Daplex Interface for the Multi-Lingual Database System

by

Beng Hock Lim
Civilian, Ministry of Defense, Republic of Singapore
B.Sc.(Hons), University of Singapore, 1974

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL

December 1986

ABSTRACT

Traditionally, the design and implementation of a conventional database system begins with the choice of a data model followed by the specification of a model-based data language. Thus, the database system is restricted to a single data model and a specific data language. An alternative to this traditional approach to database-system development is the multi-lingual database system (MLDS). This alternative approach enables the user to access and manage a large collection of databases via several data models and their corresponding data languages without the aforementioned restriction.

In this thesis we present the implementation of a functional/Daplex language interface for MLDS. Specifically, we present the implementation of an interface which translates Daplex language calls into attribute-based data language (ABDL) requests which, as the kernel language, support all other data language interfaces.

TABLE OF CONTENTS

LIST OF FIGURES

I. AN INTRODUCTION

A. THE MOTIVATION

The concept of a database system has been widely accepted by major users of computers ever since it was introduced. Many database systems have been developed and utilized by a large community of users. These database systems have been designed and implemented in a rather conventional manner - a specific data model for the database system is always selected first. then a corresponding model-based data language is specified. Some examples of these database systems are:

- IBM's Information Management System (IMS) which supports the hierarchical database model and IBM's Data Language I (DL/I)
- IBM's SQL/Data System which supports the relational model and IBM's Structured English Query Language (SQL)
- Univac's CODASYL-DML/Data System which supports the network model and Univac's CODASYL Data Manipulation Language (CODASYL-DML)
- CCA's Daplex/Data System which supports the functional model and CCA's Daplex Language

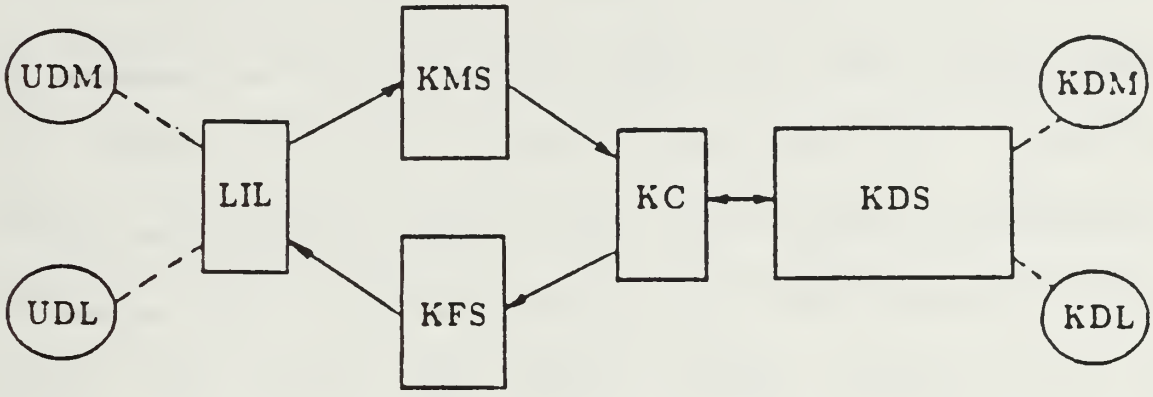
The result of this conventional approach to the design of a database system is a mono-lingual database system where the user sees and utilizes the database system with a specific data model and its model-based data language. One research effort to overcome this limitation was to introduce a new and

unconventional approach to the design and implementation of a database system, the multi-lingual database system (MLDS) [Ref. 1].

B. THE MULTI-LINGUAL DATABASE SYSTEM

The multi-lingual database system is a single database system that can execute many transactions written in different data languages and support many databases structured on different data models. The design goals of MLDS involve developing a system that is accessible via four different interfaces, the hierarchical/DL/I [Refs. 2, 3], network/DML [Ref. 4], relational/SQL [Ref. 5], and functional/Daplex [Ref. 6] models. MLDS transforms traditional database models into a single database model called the attribute-based model [Ref. 7]. The system structure of MLDS is shown in Figure 1.1.

The user data model (UDM) and the user data language (UDL) refer to the database model and language chosen by the user. The kernel data model (KDM) and the kernel data language (KDL) refer to the attribute-based model and attribute-based data language (ABDL). Users issue transactions through the language interface layer (LIL) using a chosen user data model (UDM) and written in a corresponding model-based data language (UDL). LIL then routes the user transactions to the kernel mapping system (KMS). If the user specifies that a new database is to be created, KMS transforms the database definition to a KDM-based database definition and forwards to the kernel controller (KC). KC, in turn, sends the KDM database definition to the kernel database system (KDS).



UDM : User Data Model
UDL : User Data Language
LIL : Language Interface Layer
KMS : Kernel Mapping System
KC : Kernel Controller
KFS : Kernel Formatting System
KDM : Kernel Data Model
KDL : Kernel Data Language
KDS : Kernel Database System

Figure 1.1. The Multi-lingual Database System (MLDS).

After KDS has completed its tasks of creating the database, it informs KC. KC then notifies the user via LIL. If the user specifies transactions using UDL through LIL, KMS translates the UDL transactions to the KDL transactions and sends them to KC. KC then forwards the KDL transactions to KDS for execution. Upon completion, KDS sends the results in the KDM form back to KC. KC then routes the results to the kernel formatting system (KFS) which formats

the results to the UDM form. The results in the correct UDM form are then be displayed via LIL.

The four software modules. LIL, KMS, KC, and KFS, are collectively known as the *language interface*. Four similar sets of these modules are required for the four different data model/language interfaces supported by MLDS. The hierarchical/DL/I language interface [Ref. 8], relational/SQL language interface [Ref. 9], and network/DML language interface [Ref. 10] have been implemented. The functional/Daplex language interface [Ref. 11] has been partially implemented. This thesis is a continuation on the task of implementing the functional/Daplex language interface.

C. THE KERNEL DATA MODEL AND LANGUAGE

The choice of a kernel data model and a kernel data language is the key decision in the development of a multi-lingual database system. The overriding question, when making such a choice, is whether the kernel data model and kernel data language is capable of supporting the required data-model transformations and data-language translations for the language interfaces.

The attribute-based data model proposed by Hsiao [Ref. 12], extended by Wong [Ref. 13], and studied by Rothnie [Ref. 14], along with the attribute-based data language (ABDL), defined by Banerjee [Ref. 15], have been shown to be acceptable candidates for the kernel data model and kernel data language, respectively.

The determination of a kernel data model and kernel data language is important for MLDS because no matter how multi-lingual MLDS may be, if the underlying database system (i.e., KDS) is slow and inefficient, then the interfaces may be rendered useless and untimely. Hence, it is important that the kernel data model and kernel language be supported by a high-performance and great-capacity database system. Currently, only the attribute-based data model and the attribute-based data language are supported by such a system. This system is the multi-backend database system (MBDS) [Ref. 16].

D. THE MULTI-BACKEND DATABASE SYSTEM

The multi-backend database system (MBDS) has been designed to overcome the performance problems and upgrade issues related to the traditional approach of database system design. This goal is realized through the utilization of multiple backends connected in a parallel fashion. These backends have identical hardware, replicated software, and their own disk systems. In a multiple backend-configuration, there is a backend controller, which is responsible for supervising the execution of database transactions and for interfacing with the hosts and users. The backends perform the database operations with the database stored on the disk system of the backends. The controller and backends are connected by a communication bus. Users access the system through either the hosts or the controller directly (see Figure 1.2).

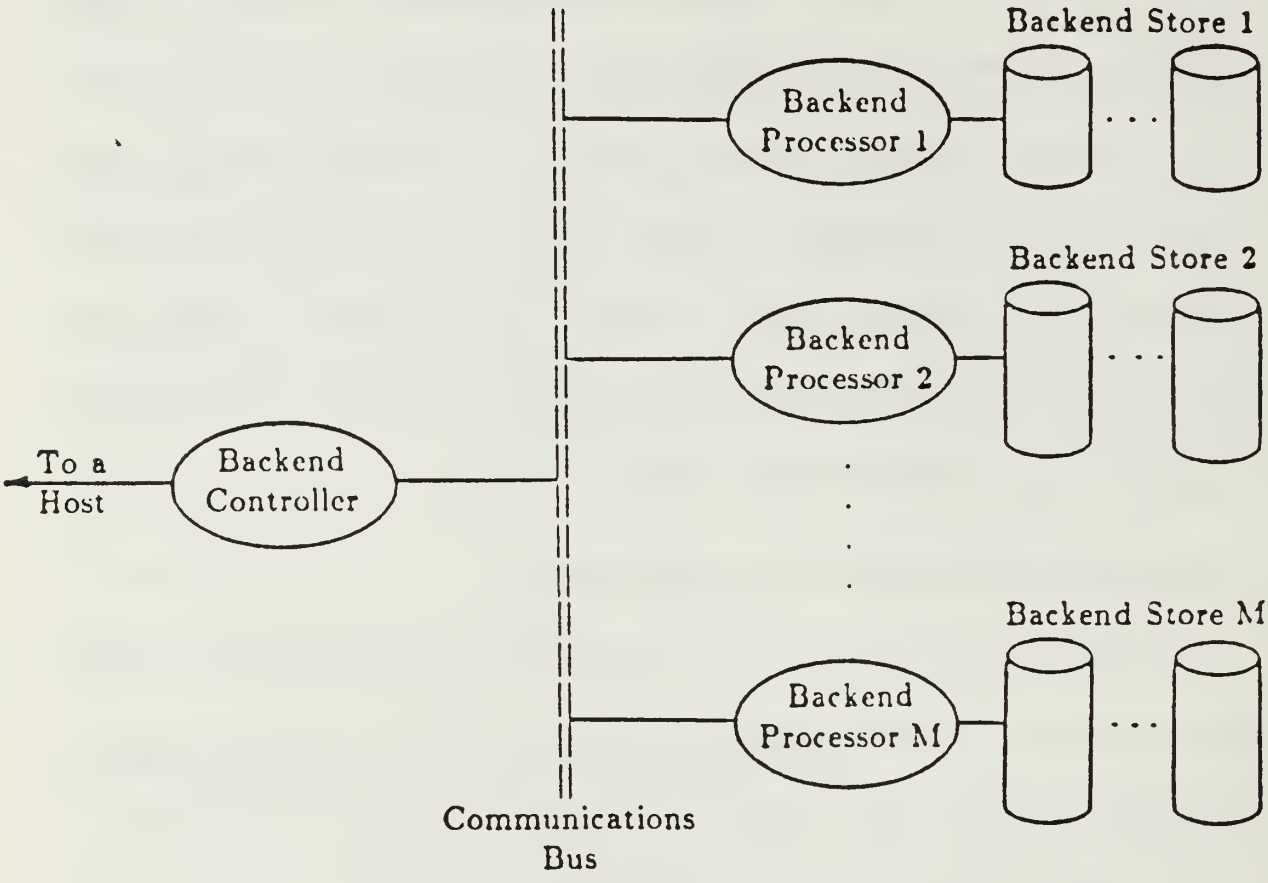


Figure 1.2. The Multi-Backend Database System.

Performance gains are realized by increasing the number of backends. If the size of the database and the size of the responses to the transactions remain constant, then MBDS produces a reciprocal decrease in the response times for the user transactions when the number of backends is increased. On the other hand, if the number of backends is increased proportionally with the increase in databases and responses, then MBDS produces invariant response times for the same transactions. A more detailed discussion of MBDS is found in [Ref. 16].

E. THE ORGANIZATION OF THE THESIS

The remainder of this thesis is organized into five chapters and five appendices. In Chapter II, we present the background materials. This includes an overview of the source data language, Daplex and the target data language, ABDL. In Chapter III, we describe the implementation strategy, the program modules and the data structures used in the functional/Daplex language interface. In Chapter IV, we present additions and modifications to the language interface layer from the first implementation [Ref. 11]. In Chapter V, we describe the mapping process of the kernel mapping system. Finally, in Chapter VI, we conclude the thesis. Of the appendices, Appendix A contains the data structures used for the interface; Appendix B and C contain the program listings for the LIL and KMS modules; And, Appendix D lists the grammar rules used in our implementation of the Daplex interface.

II. THE BACKGROUND MATERIALS

A. THE SOURCE DATA LANGUAGE - DAPLEX

Daplex is a database language that was created by David W. Shipman [Ref. 17] in 1979 while working at the Computer Corporation of America (CCA) and the Massachusetts Institute of Technology (MIT) [Ref. 18]. Its foundation is in what Shipman and Gray [Ref. 18] call the functional data model. One of Shipman's goals for Daplex is to provide a *conceptually natural* database interface language. That is, the Daplex constructs used to model real-world situations are intended to closely match the conceptual constructs a human being might employ when thinking about those situations. Such conceptual naturalness, to the extent it has been achieved, presumably simplifies the process of writing and understanding Daplex requests, since the translation between the user's mental representation and its formal expression in Daplex is more direct.

Gray notes that Shipman developed his concepts from the semantic net data abstraction used in artificial intelligence. The semantic net is a structure that represents associations between objects. For each object of a given type, there is a corresponding collection of functions which are applicable to it; some of these provide simple values, but the results of others are found by following arcs in the net, which connect the object to other objects of various types. Functions can be

applied in turn to these objects, thus determining a network of associations. Consequently, Shipman's Daplex relies on functions and functional composition to derive actual values.

A sample database, *univ,* is used for examples throughout this thesis. The database schema is presented in Figure 2.1 and a graphical representation of the database is presented in Figure 2.2. This database follows closely the sample database. *university,* in the Daplex User's Manual [Ref. 19] except that some identifier names have been abbreviated. The abbreviation is necessary as the attribute-based data language requires the identifier name to be of eight or less characters in length and the use of underscore in an identifier name is not permitted.

The Daplex database language consists of two parts, the data definition language (DDL) and the data manipulation language (DML) . DDL defines the logical types and structures of information in databases and the constraints that specify which values are legal. DML allows a database user to create, delete, modify, and retrieve information in the databases.

1. The Data Definition Language

Daplex database definitions are syntactically similar to programming language declarations. The basic construct of DDL is the database declaration. A data declaration consists of data declarations and constraints. The data declarations define the types and structures of the database. The constraints define the set of legal values of the database. The basic format of a database

```
DATABASE univ IS

  TYPE person;
  SUBTYPE employee;
  SUBTYPE supstaff;
  SUBTYPE faculty;
  SUBTYPE student;
  SUBTYPE graduate;
  SUBTYPE undrgrad;
  TYPE course;
  TYPE dept;
  TYPE enroll;
  TYPE rankname IS (assistant,associate,full);
  TYPE semester IS (fall, spring, summer);
  TYPE ptgrade IS RANGE 0.0 .. 4.0;

  TYPE person IS
   ENTITY
    name      : STRING (1 .. 25);
    ssn       : STRING (1 .. 9) := "000000000";
   END ENTITY;

  SUBTYPE employee IS person
   ENTITY
    homeaddr: STRING (1 .. 50);
    office    : STRING (1 .. 8);
    phones    : SET OF STRING (1 .. 7);
    salary    : FLOAT;
    depdents  : INTEGER RANGE 0 .. 10 ;
   END ENTITY;

  SUBTYPE supstaff IS employee
   ENTITY
    supvisor  : employee WITHNULL;
    fulltime  : BOOLEAN;
   END ENTITY;

  SUBTYPE faculty IS employee
   ENTITY
    rank      : rankname;
    teaching  : SET OF course;
    tenure    : BOOLEAN := FALSE;
    fdept     : dept;
   END ENTITY;
```

Figure 2.1. The Sample Database, univ. (continued)

```
  SUBTYPE student IS person
   ENTITY
    advisor    : faculty WITHNULL;
    major      : dept;
    enrolls    : SET OF enroll;
   END ENTITY;

  SUBTYPE graduate IS student
   ENTITY
    advcomm : SET OF faculty;
   END ENTITY;

  SUBTYPE undrgrad IS student
   ENTITY
    gpa        : ptgrade := 0.0;
    year       : INTEGER RANGE 1 .. 4 := 1;
   END ENTITY;

  TYPE course IS
   ENTITY
    title      : STRING (1 .. 10);
    fdept      : dept;
    fsemster   : semester;
    credits    : INTEGER;
   END ENTITY;

  TYPE dept IS
   ENTITY
    name       : STRING (1 .. 20);
    head       : faculty WITHNULL;
   END ENTITY;

  TYPE enroll IS
   ENTITY
    class      : course;
    grade      : ptgrade;
   END ENTITY;

  UNIQUE ssn WITHIN person;
  UNIQUE name WITHIN dept;
  UNIQUE title, fsemster WITHIN course;
  OVERLAP graduate WITH faculty;

END univ;
```

Figure 2.1. The Sample Database, univ.

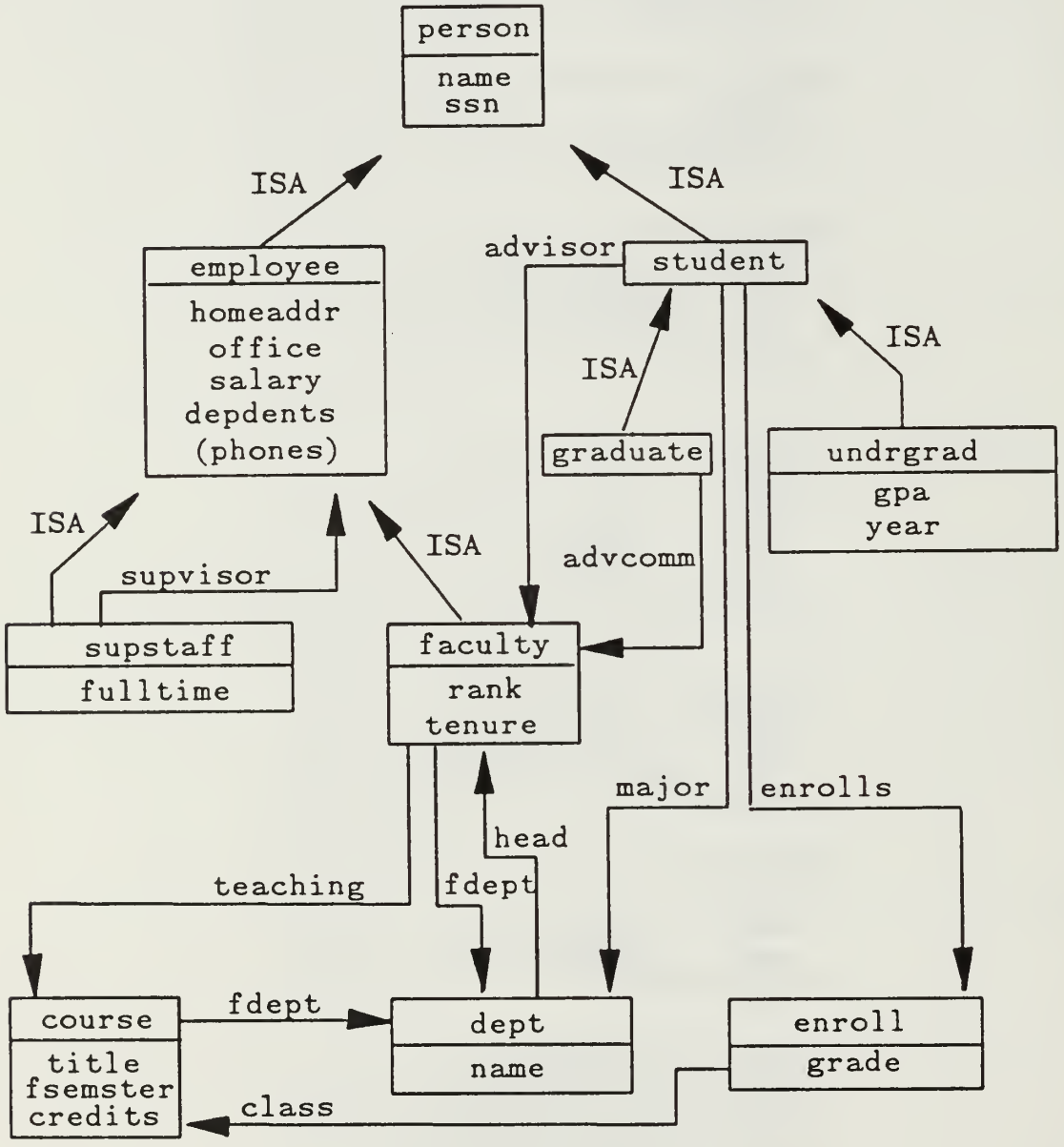


Figure 2.2. A Graphical Representation of the "univ" Database Schema.

declaration is given in Figure 2.3. Figure 2.1 is a complete example of a database declaration. In this figure, the square brackets denote that the content enclosed between them is optional. This syntactic convention is used throughout this thesis.

```
DATABASE db-name IS
    [ non-entity-type-declarations ]
    entity-type-declarations
    [ entity-type-constraints ]
END [ db-name ];
```

Figure 2.3. The Database-Declaration Format.

The db-name refers to the unique name of the database being declared. The non-entity-type declarations are declarations of string types, scalar types, and numeric constants. The entity-type-declarations are declarations of entity types and subtypes, their functions and generalization hierarchies. The entity-type-constraints define those properties of the declared entity types and subtypes that must remain invariant under any operations on values of those types. The data declarations and constraints can be intermixed in any order. However, all types must be completely or partially declared before the name of the type can appear in another declaration.

An entity-type declaration declares a new entity type. The two formats of entity-type declarations are shown in Figure 2.4. The first format is for a

complete entity-type declaration. The entity-type-name is an identifier that is the name of the entity being declared. The name must be unique among all type and subtype names within the database. The function-names are lists of one or more identifiers, separated by commas, that are the names of the functions that can be applied to the entity type being declared. If there is more than one identifier in the list, all function names on the list share the same function-type. The function-type may be strings, scalars (integer, floating-point, or enumeration type), entities, nonentities or sets of any of the above types. The second format is for a partial entity-type declaration. A partial entity-type declaration introduces only the type name to make it available as a function type for another entity type declaration. A function type must be declared, completely or partially, before it

---

```
(1) TYPE entity-type-name IS
      ENTITY
        [ function-names-1 : function-type;
          function-names-2 : function-type;
                  .
                  .
                  .
          function-names-n : function-type; ]
      END ENTITY;


(2) TYPE entity-type-name;
```

Figure 2.4. The Entity-Type-Declaration Formats.

---

can be referenced. The reference of entity types in this manner provides the operational link in implementing the user-defined relationships of the functional model in Daplex. Whenever a partial declaration appears in an entity-type declaration, the complete declaration of that entity type must appear later in the same database declaration.

The relations among the objects in Daplex are reflected in the generalization hierarchies. All types in the generalization hierarchy are entity types or entity subtypes. An example of the Person generalization hierarchy is presented in Figure 2.5. The example shows that the entity type, Person, forms

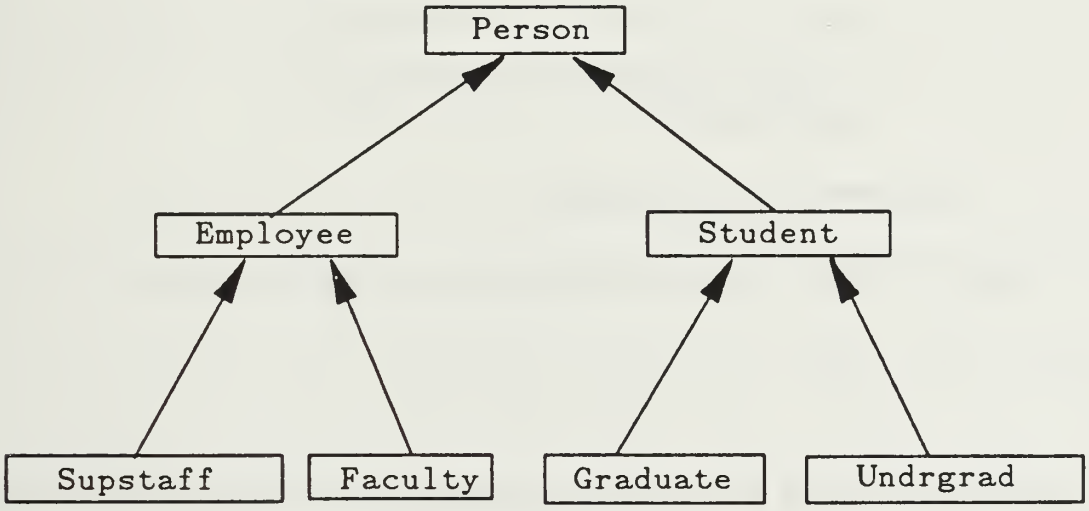


Figure 2.5. An Example of Generalization Hierarchy.

the root of a tree of built-in relationships. The nodes of the tree are entity subtypes. The names of subtypes must be unique. Subtypes are descendants of the root type or other subtypes. As the hierarchy is traversed downward from level to level, each subtype inherits all of the functions of its supertypes. Supertypes are ancestors of subtypes. The highest level ancestor (Person) is the root type. The formats for generalization-hierarchy declarations are depicted in Figure 2.6.

```
(1) SUBTYPE subtype-name IS supertype-names
      ENTITY
         [ function-names-1 : function-type;
           function-names-2 : function-type;
                    .
                    .
                    .
           function-names-n : function-type; ]
      END ENTITY;

(2) SUBTYPE subtype-name;
```

Figure 2.6. The Generalization-Hierarchy-Declaration Formats.

The subtype-name is an unique identifier of the subtype being declared. The supertype-names is a list of one or more names of entity types and subtypes of built-in relationships completely declared previously in an entity-type declaration or generalization-hierarchy declaration. The function-names and the function-types are the same as for an entity-type declaration. The two formats

also correspond directly to the entity type declaration formats. One important point to remember is that the complete declaration of all subtypes creates a built-in relationship.

Nonentity-type declarations describe the primitive objects that declare data types other than entities. There are three nonentity types: the base type, the subtype and the derived type. Their formats are shown in Figure 2.7.

---

base type : TYPE type-name IS type-definition:

subtype : SUBTYPE subtype-name IS prev-name
[scalar-type-constraints]:

derived type : TYPE type-name IS NEW prev-name
[scalar-type-constraints]:

Figure 2.7. The Nonentity-Type-Declarations Format.

The type-definition declares the data types of the nonentities. They may be user-defined or predefined. The predefined types are STRING, INTEGER, FLOAT, and BOOLEAN. User-defined types are strings, scalars (integer, floating-point, enumeration, and boolean), and numeric constants. The prev-name refers to name of previously declared type or subtype. The scalar-type-constraint may be used to restrict the value of the nonentity type being declared. An example of a scalar-type-constraint is the restriction of "ptgrade" within the range from "0.0" to "0.4"

There are two types of constraints on entity types in Daplex: the overlap constraint and uniqueness constraint. An overlap constraint determines when an entity may legally belong to two or more terminal entity subtypes. A terminal entity subtype is a leaf in the generalization hierarchy. Terminal types are disjoint unless they are connected with the overlap constraint. The format of overlap-constraint declaration is shown in Figure 2.8. The entity-type-names is a list of one or more identifiers (separated by commas) of previously declared entity subtypes. An overlap constraint specifies that any database entity belonging to a subtype identified in the first list also may belong to each of the subtypes identified in the second list.

OVERLAP entity-type-names WITH entity-type-names:

Figure 2.8. The Overlap-Constraint Format.

The uniqueness constraint specifies, for a particular entity type or subtype, a collection of functions whose values are unique for all entities in a database belonging to that type or subtype. The format for uniqueness-constraint is given in Figure 2.9. The entity-type-name is the name of a previously declared entity type or subtype. A restriction for values of functions is that uniqueness constraints only apply to values directly. Functions that derive their values from relationships with entities or nonentities are precluded from forming uniqueness constraints.

UNIQUE function-names WITHIN entity-type-name:

Figure 2.9. The Uniqueness-Constraint Format.

The Daplex DDL is to be discussed again in Chapter V. where the transformations between the Daplex and the ABDL structures are presented.

2. The Data Manipulation Language

The Daplex data manipulation language (DML) allows users to retrieve and update entities and function values in the database. The FOR EACH statement is the basis for Daplex data retrieval. It is used to specify the set of entities or values that are to be accessed and to loop over that set. The format of the FOR EACH statement is shown in Figure 2.10. The loop-label is an optional

```
[ loop-label: ] FOR [ EACH ] loop-parameter
        IN set-expression
        [ WHERE boolean-expression ]
        [ BY order-clauses ]
[ LOOP ]
    loop-body
END [ LOOP ];
```

Figure 2.10. The FOR EACH Statement Format.

label for the statement. The loop-parameter is an identifier. On each iteration of the loop, loop-parameter is assigned a member of the set of database values. The set-expression specifies a set of unique string, scalar. or entity values. The boolean-expression is any expression that yields a boolean value. It restricts the members of set-expression over which the loop iterates. Order-clause is a list of one or more clauses that define the order of iteration through the set. The loop-body may include a sequence of one or more DML statements that are executed once for each iteration of the loop.

For printing and formatting the display of query results. the PROCEDURE-CALL statement is used. There are several formats available in the original Daplex data language. In our implementation, only the PRINT and PRINT_LINE statement formats, shown in Figure 2.11. are implemented. These statements print the values of each of the expressions using default formats of the expressions. The difference between the two statements is that PRINT_LINE outputs a carriage return after all of the expression values are printed.

---

(1) PRINT (expressions);

(2) PRINT_LINE (expressions);

Figure 2.11. The PRINT and PRINT_LINE Statement Formats.

For database updates, there are the ASSIGNMENT, CREATE, INCLUDE, EXCLUDE, DESTROY and MOVE statements. The formats of these statements are listed in Figure 2.12. The ASSIGNMENT statement is used to

---

(1) ASSIGNMENT statement:
single-valued-function-expression := expression;

(2) CREATE statement:
CREATE NEW entity-type-names
[(function-name-1 => expression-1,
function-name-2 => expression-2,
.
.
.
function-name-m => expression-m)];

(3) INCLUDE statement:
INCLUDE expression INTO set-valued-function-expression;

(4) EXCLUDE statement:
EXCLUDE expression FROM set-valued-function-expression;

(5) DESTROY statement:
DESTROY (entity-valued-expression);

(6) MOVE statement:
MOVE entity-valued-expression
[FROM entity-type-names]
[INTO entity-type-names
[(function-name-1 => expression-1,
function-name-2 => expression-2,
.
.
.
function-name-m => expression-m)]];

Figure 2.12. The Database-Update-Statement Formats.

---

assign or modify a value to a single-valued function. The CREATE statement is used to create a new database entity. The INCLUDE statement adds either a single value or a set of values to a set-valued function. The EXCLUDE statement is used to remove a single value or a set of values from a set-valued function. The DESTROY statement removes an entity from the database. The MOVE statement changes the subtypes to which an entity belongs. The effect is the removal of the entity from each subtype in the FROM list and the addition of the entity to each subtype in the INTO list.

B. THE TARGET DATA LANGUAGE - ABDL

1. The Attribute-based Data Model

The data structures of the attribute-based data model include: database, file, record, attribute-value pair, keyword, attribute-value range, directory keyword, non-directory keyword, directory, record body, predicates, and query. Informally, a *database* consists of a collection of files. Each *file* contains a group of records characterized by a unique set of directory keywords. An example of a record for a "Person" file is shown in Figure 2.13. A *record* is composed of two

---

(<FILE, Person>, <NAME, Charlie Brown>, <SSN, 123456789>,
{ Cartoon character })

Figure 2.13. An Example of a Record.

---

parts. The first part of a record is a collection of *attribute-value pairs* or *keywords*. The *attribute* of an attribute-value pair defines the specific quality or the certain characteristics of the value. In our example, the attribute-value pair, <NAME, Charlie Brown>, has an attribute "NAME" and the value of the attribute is "Charlie Brown" Each record can only have one value associated with a corresponding attribute in the attribute-value pair. Further, no two attribute-value pairs have the same attribute in a record, i.e., all attributes must be unique in a record. Certain attribute-value pairs of a record are called *directory keywords* since their attribute values or attribute-value ranges are kept in a directory for identifying the records (files). <FILE, Person> in Figure 2.13 is an example of a directory keyword. Those attribute-value pairs that are not kept in the directory are called *non-directory keywords*. The second part of the record is the *record body* which contains only textual information. The record body in our record is the "Cartoon character" enclosed within a pair of curly brackets.

The records of a database may be identified by keyword predicates. A *keyword predicate* is a 3-tuple consisting of a directory attribute, a relational operator, and an attribute value, e.g., (NAME = Charlie Brown). A query combines keyword predicates in disjunctive normal form. An example of a query is given in Figure 2.14. The query will be satisfied by all records of the Person file where the value of the attribute NAME is "Charlie Brown" or "Beetle Bailey" use parenthesis for bracketing conjunctions in a query.

((FILE = Person) and (NAME = Charlie Brown)) or
((FILE = Person) and (NAME = Beetle Bailey))

Figure 2.14. An example of a Query.

2. The Attribute-based Data Language

The attribute-based data language (ABDL) supports five primary database operations: INSERT, DELETE, UPDATE, RETRIEVE, and RETRIEVE-COMMON. A *request* in ABDL is a primary operation with a qualification. A *qualification* specifies the part of the database that is to be operated on. Two or more requests may be grouped together to form a *transaction.*

The INSERT request inserts a new record into the database. The qualification of an INSERT request is a list of keywords with or without a record body. An example of a INSERT request is shown in Figure 2.15. This request inserts a new record to the Person file where the value of the attribute NAME is

INSERT (<FILE, Person>, <NAME, Charlie Brown>
<SSN, 123456789>, <AGE, 35>)

Figure 2.15. An Example of INSERT Request.

Charlie Brown, the value of the attribute SSN is 123456789, and the value of the attribute AGE is 35.

A DELETE request removes one or more records from a database. The qualification of a DELETE request is a query. An example is shown in Figure 2.16 to illustrate the DELETE request. This request removes all of the records from the Person file whose age is greater than 60.

DELETE ((FILE = Person) and (AGE > 60))

Figure 2.16. An Example of DELETE Request.

The UPDATE request modifies records of the database. The qualification of an UPDATE request consists of two parts, the query and the modifier. The query specifies which records of the database are to be modified. The modifier specifies how the records being modified are to be updated. An example of the UPDATE request is shown in Figure 2.17. This request modifies all of the records in the Person file by incrementing the age by one. The query is (FILE = Person) and the modifier is (AGE = AGE + 1).

UPDATE (FILE = Person) (AGE = AGE + 1)

Figure 2.17. An Example of UPDATE Request.

A RETRIEVE request retrieves records from the database. The qualification of a retrieve request consists of a query, a target-list, and an optional by-clause. The query specifies which records are to be retrieved. The target-list consists of a list of output attributes. It may also consist of an aggregate operation. i.e., AVG, COUNT, SUM, MIN, MAX, on one or more output attributes. The optional by-clause may be used to group records when an aggregate operation is specified. Figure 2.18 is an example of the RETRIEVE request. This request lists the SSN, NAME and AGE of all records in the Person file whose AGE value is greater than seventeen. The query is (FILE = Person) and (AGE > 17), the target-list is (SSN, NAME, AGE), and the by-clause is by SSN.

RETRIEVE ((FILE = Person) and (AGE > 17))
(SSN, NAME, AGE) by SSN

Figure 2.18. An Example of RETRIEVE Request.

The last request, RETRIEVE-COMMON, is used to merge two files by common attribute-values. Logically, the RETRIEVE-COMMON request can be considered as a transaction of two retrieve requests that are processed serially in the general form shown in Figure 2.19. The common attributes are attribute-1 (associated with the first retrieve request) and attribute-2 (associated with the second retrieve request).

```
RETRIEVE (query-1) (target-list-1)
COMMON  (attribute-1. attribute-2)
RETRIEVE (query-2) (target-list-2)
```

Figure 2.19. The General Form of RETRIEVE-COMMON Request.

In Figure 2.20. an example of RETRIEVE-COMMON from a population census example illustrates this request. This example finds all of the records in the CanadaCensus file with population greater than 100.000. finds all the records in the USCensus file with population greater than 100.000. identifies records of respective files whose population figures are common. and returns the two city names whose cities have the same population figures. ABDL provides five seemingly simple database operations, which are nevertheless capable of supporting complex and comprehensive transactions.

```
RETRIEVE ((FILE = CanadaCensus) and
        (POPULATION >= 100000))(CITY)
COMMON (POPULATION, POPULATION)
RETRIEVE ((FILE = USCensus) and
        (POPULATION >= 100000))(CITY)
```

Figure 2.20. An Example of RETRIEVE-COMMON Request.

III. THE IMPLEMENTATION PROCESS

A. THE DESIGN GOALS

During the early stages of the design of MLDS. we have decided that there should not be any changes to the kernel data model and the kernel data language, i.e., the attribute-based data model and ABDL. Instead, our implementation resides entirely in the host computer system. All user transactions in Daplex are processed in the Daplex interface and transformed into ABDL transactions format before being forwarded by KC to KDS for processing.

In addition. we intend to make our interface transparent to the user. For example, a user with previous experience in Daplex could log onto our system, issue a Daplex request and receive result data in the functional format. The user requires no training in ABDL procedures prior to utilizing the system.

B. THE IMPLEMENTATION STRATEGY

Due to the large size of the MLDS project and the time constraint on the thesis students involved, a suitable implementation strategy must be employed. We have predicated our choice of the strategy on minimizing the "software-crisis" as explained by Boehm [Ref. 20]. The strategy we have decided upon is the level-by-level, top-down approach. The system is initially thought of as a "black box" that accepts Daplex transactions and then returns the appropriate

results. The "black box" is then decomposed into its four modules (i.e., LIL, KMS, KC, and KFS). These modules, in turn, are further decomposed into the necessary functions and procedures to accomplish the appropriate tasks.

The design of the language interface uses a Systems Specification Language (SSL) [Ref. 21] extensively. SSL has permitted us to approach the design from a very high-level, abstract perspective. Furthermore, SSL has allowed us to make an easy transition from the design phase to the implementation phase.

We have used the C programming language [Ref. 22] to implement the language interface. The greatest advantage of using C is the programming environment that it resides (i.e., the UNIX operating system). This environment has permitted us to partition the Daplex interface and then manage the parts in an effective and efficient manner. One disadvantage with using C is its poor error diagnostics that makes debugging difficult. There is an on-line debugger available for use with C in UNIX for debugging. We have avoided this option and instead used conditional compilation and diagnostic print statements to aid in the debugging process. Another criticism on C is that programs written in C are often cryptic. A C program is usually less readable than a program written in other conventional programming languages. We have made every effort to ensure that the C code we have written to be easy to read and comprehend. For instance, we often write the code with extra lines to avoid shorthand notations available in C. These extra lines have made the difference between comprehensible code and cryptic notations. We have also

intentionally minimized the interaction between procedures to ease the burden of maintainability.

C. THE DATA STRUCTURE

The Daplex language interface has been developed as a single user system that at some point will be updated to a multi-user system. Two different concepts of the data are used in the language interface : (1) Data shared by all users, and (2) Data specific to each user. The reader must realize that the data structures used in our interface and described below have been deliberately made generic. Hence. these same structures support not only our Daplex interface, but the other language interfaces as well. i.e.. DL/I. CODASYL-DML. and SQL.

1. Data Shared by All Users

The data structures that are shared by all users are the database schemas defined by the users thus far. In our case, these are the functional schemas, consisting of entities (types and subtypes) and the relationships (functions) between the entities. These are not only shared by all users, but also shared by the four modules of the MLDS, i.e., LIL, KMS, KC, and KFS. Figure 3.1 depicts the first data structure used to maintain data.

It is important to note that this structure is represented as a union. Hence, it is generic in the sense that a user can utilize this structure to support SQL, DL/I, CODASYL-DML, or Daplex needs. However, we concentrate

```
union dbid_node
  {
    struct rel_dbid_node    *dn_rel;
    struct hie_dbid_node    *dn_hie;
    struct net_dbid_node    *dn_net;
    struct fun_dbid_node    *dn_fun;
  }
```

Figure 3.1. The dbid_node Data Structure.

only on the functional model. In this regard, the fourth field of this structure points to a record that contains information about a functional database. Figure 3.2 illustrates the structure of this record.

The fdn_name is simply a character array containing the name of the functional database. The fdn_nonentptr field holds a pointer to the base-type nonentity node, and the fdn_num_nonent is an integer value that represents the number of these nodes in the database. The fdn_entptr points to the entity node, and as before the field that immediately follows contains an integer value representing the number of such nodes. The fdn_subptr is a pointer to the generalized entity subtype node and the integer field following it keeps the number of these generalized subtype nodes in the database. The fdn_nonsubptr is a pointer to the nonentity subtypes and the integer following it keeps the number of such nodes in the database. The fdn_nonderptr points to the nonentity derived type node and the number of these nodes in the database is kept in the next field.

```
struct fun_dbid_node
  {
    char    fdn_name[DBNLength + 1];
    struct  ent_non_node      *fdn_nonentity;
    int     fdn_num_nonent:
    struct  ent_node          *fdn_entity;
    int     fdn_num_ent:
    struct  gen_sub_node      *fdn_subptr;
    int     fdn_num_gen;
    struct  sub_non_node      *fdn_nonsubptr;
    int     fdn_num_nonsub;
    struct  der_non_node      *fdn_nonderptr;
    int     fdn_num_der:
    struct  overlap_node      *fdn_ovrptr;
    int     fdn_num_ovr;
    struct  fun_dbid_node     *fdn_next_db:
  };
```

Figure 3.2. The fun_dbid_node Data Structure.

The fdn_ovrptr and fdn_num_ovr are new fields added to the fun_dbid_node structure. The fdn_ovrptr is a pointer to the structure overlap_node which contains the overlapping constraints of the database and the fdn_num_ovr keeps track of the number of such constraints. Finally, the last field points to the next functional database node.

Figure 3.3 depicts the entity node structure. The first field of this structure is a character array which holds the name of the entity, and the second field is an integer that is used for keeping track of the last unique number assigned to each entity type in the database. The third field is an integer

```
struct ent_node
  {
    char     en_name[ENLength + 1];
    int      en_last_ent_id;
    int      en_num_funct;
    int      en_terminal;
    struct   function_node      *en_ftnptr;
    struct   ent_node           *en_next_ent;
  };
```

Figure 3.3. The ent_node Data Structure.

representation of the number of functions associated with the entity that this node represents. For instance, the "person" entity has two functions associated with it, "name" and "ssn" The fourth field is an integer representation of a boolean function for indicating whether the entity is a terminal type. An entity type is a terminal type if it is not a supertype to any subtype. The en_ftnptr is a pointer to the function node associated with this entity node. The last field points to the next entity node in the database.

The structure of the gen_sub_node is shown in Figure 3.4. The first field, similar to previous structures, holds the name of the generalized entity subtypes. An example applied to the *univ* data base is "employee" The gsn_num_funct field holds the number of functions associated with the entity subtype, and the gsn_terminal field is an integer representation of a boolean function and holds a "1" if the generalized subtype entity is not a supertype. The gsn_entptr field

```
struct gen_sub_node
  {
    char     gsn_name[ENLength + 1];
    int      gsn_num_funct;
    int      gsn_terminal:
    struct   ent_node_list       *gsn_entptr;
    int      gsn_num_ent:
    struct   function_node       *gsn_ftnptr:
    struct   sub_node_list       *gsn_subptr;
    int      gsn_num_sub;
    struct   gen_sub_node        *gsn_next_genptr;
  };
```

Figure 3.4. The gen_sub_node Data Structure.

holds a pointer to its supertype which is of entity type. In the case of "employee" , the supertype is "person" The next field indicates the number of entity supertypes. The gsn_ftnptr field holds a pointer to a function associated with the generalized subtype, for instance, "salary" The gsn_subptr field holds a pointer to the subtype supertype. For example, the supertype for the subtype "supstaff" is "employee" of the number of such subtype supertypes. The final field simply points to the next gen_sub_node.

The ent_non_node record is shown in Figure 3.5, and contains information about each nonentity base-type in the database. The first field of the record holds the name of the nonentity node, for example, "rankname" holds the character that indicates the type of nonentity node, either "i" , integer; "e" , enumeration; "f" , floating point; "s" , string; "b" , boolean. The next field

```
struct  ent_non_node
   {
     char     enn_name[ENLength + 1];
     char     enn_type;
     int      enn_total_length;
     int      enn_range;
     int      enn_num_values;
     struct   ent_value          *enn_value;
     int      enn_constant;
     struct   ent_non_node       *enn_next_node;
   };
```

Figure 3.5. The ent_non_node Data Structure.

contains an integer that indicates the maximum length of the base-type value. The enn_range field contains an integer representation of a boolean value, a "1" or "0" , that indicates whether or not there is a range associated with the nonentity node. For example, from Figure 2.1, the nonentity "ptgrade" has a range of from 0.0 to 4.0, while "rankname" is without a range. The enn_num_values field contains an integer that represents the number of different values that the nonentity can assume. The next field contains a pointer to the actual value of the base-type. As an example, "rankname" can assume three values, the enn_num_values would have a value "3" stored and the enn_value would point to a link list which contains the three values "assistant" , "associate" , and "full" If a value range is associated with the nonentity node, indicated by enn_range field, then the enn_num_values would be two and the

enn_value will point to a linked list which contains the lower and upper bounds of the range. The enn_constant field contains an integer representation of a boolean value that indicates if the actual value of the base-type is a constant. There are no constants in the *univ* database, but, as an example, the value of the base-type could assume the constant value of pi (3.14159265). The last field contains a pointer to the next nonentity node.

The sub_non_node is shown in Figure 3.6. This structure is almost identical in form and similar in purpose to the ent_non_node of Figure 3.3. The main difference between the two structures is that the ent_non_node is for a base-type nonentity and the sub_non_node is for a subtype nonentity. The difference in form between the two structures is the absence of constants in the sub_non_node. Maintaining two separate constant lists would be redundant, hence the constants are found only in the ent_non_node.

```
struct sub_non_node
   {
     char     snn_name[ENLength + 1];
     char     snn_type;
     int      snn_total_length;
     int      snn_range;
     int      snn_num_values;
     struct   ent_value          *snn_value;
     struct   sub_non_node       *snn_next_node;
   };
```

Figure 3.6. The sub_non_node Data Structure.

The next node, similar to both the ent_non_node and the sub_non_node, is the der_non_node. shown in Figure 3.7. The der_non_node is identical in structure to the sub_non_node and differs in function in that it applies to the derived nonentity subtypes.

```
struct  der_non_node
   {
     char     dnn_name[ENLength + 1];
     char     dnn_type;
     int      dnn_total_length;
     int      dnn_range;
     int      dnn_num_values;
     struct   ent_value          *dnn_value;
     struct   der_non_node       *dnn_next_node;
   };
```

Figure 3.7. The der_non_node Data Structure.

The structure of the overlap_node is shown in Figure 3.8. The first field refers to the name of the base type for the overlapping entities. For example, in the case of overlapping "graduate" with "faculty" , the base_type_name is "person" The second field holds a pointer to the list of terminal subtypes that are overlapped. The third field keeps a count of overlapping subtypes in the list. The last field holds a pointer to the next overlap_node.

The structure of the function_node is shown in Figure 3.9. The function_node defines the structures for each function-type declaration. The first field of the function_node is a character array which holds the name of the

```
struct  overlap_node
   {
     char     base_type_name[ENLength+1];
     struct   sub_node_list        *snlptr;
     int      num_sub_node;
     struct   overlap_node         *next:
   };
```

Figure 3.8. The overlap_node Data Structure.

function. The second field is a character that holds the function type. The value of this field could be "f" , "i" , "s" , "b" or "e" corresponding to the types float, integer, string, boolean, or entity respectively. The fn_range field indicates

```
struct  function_node
   {
     char     fn_name[ENLength+1];
     char     fn_type;
     int      fn_set;
     int      fn_range;
     int      fn_total_length;
     int      fn_num_value;
     struct   ent_value            *fn_value;
     struct   ent_node             *fn_entptr;
     struct   gen_sub_node         *fn_subptr;
     struct   ent_non_node         *fn_nonentptr;
     struct   sub_non_node         *fn_nonsubptr;
     struct   der_non_node         *fn_nonderptr;
     int      fn_entnull;
     int      fn_unique;
     struct   function_node        *next;
   };
```

Figure 3.9. The function_node Data Structure.

whether the function has a range of values, and the fn_set field indicates whether the function is a set-valued function. A "0" in this field would indicate that the function is single-valued.

The fn_total_length field indicates the maximum length, the fn_num_value field indicates the number of values, and the fn_value field would hold a pointer to the actual values. The next five fields hold pointers to the type to which a particular function belongs. For example, the function "head" in the entity "dept" would have the fn_subptr pointing to the generalized subtype "faculty" and the other four type field pointers will be null. When the function is an entity-valued function, the fn_entnull field would indicate whether the function may have a null value. The fn_unique field indicates whether an uniqueness constraint is applicable to the function. It is always initialized to "0" and set to "1" only when an uniqueness constraint is specified. For example, the "ssn" function in the "person" entity would have this field set to "1" The final field simply contains a pointer to the next function.

The ent_node_list and sub_node_list data structures are shown in Figure 3.10. These two structures are used for keeping linked list of pointers to entity nodes and generalized subnodes , respectively.

The data structure of ent_value is shown in Figure 3.11. It is used for keeping a linked list of entity values. As the length of an entity value is not a constant, the ev_value field is just a character pointer. Exact size of memory could be allocated at the time when an entity value is to be stored.

```
struct ent_node_list
   {
     struct   ent_node          *entptr;
     struct   ent_node_list     *next;
   };

struct sub_node_list
   {
     struct   gen_sub_node      *subptr;
     struct   sub_node_list     *next;
   };
```

Figure 3.10. The ent_node_list and sub_node_list Data Structures.

```
struct ent_value
   {
     char     *ev_value;
     struct   ent_value     *next;
   };
```

Figure 3.11. The ent_value Data Structures.

2. Data Specific to Each User

This category of data represents information needed to support each user's particular interface needs. The data structures used to accomplish this can be thought of as forming a hierarchy. At the root of this hierarchy is the data structure, user_info, that maintains information on all of the current users of a particular language interface (see Figure 3.12). The user_info data structure holds the ID of the user, a union that describes a particular interface, and a pointer to the next user. The union field is of particular interest to us. As

```
struct  user_info
  {
    char     uid[UIDLength + 1];
    union    li_info        li_type:
    struct   user_info      *next_user;
  }
```

Figure 3.12. The user_info Data Structure.

noted earlier. a union serves as a generic data structure. In this case. the union can hold the data for a user accessing either a CODASYL-DML language interface layer(LIL), a DL/I LIL. an SQL LIL. or a Daplex LIL. The li_info union is shown in Figure 3.13.

We are only interested in the data structures containing user information which relates to the Daplex language interface in this section. The structure used is referred to as dap_info and is depicted in Figure 3.14. The

```
union  li_info
  {
    struct  sql_info          li_sql;
    struct  dli_info          li_dli;
    struct  dml_info          li_dml;
    struct  dap_info          li_dap;
  }
```

Figure 3.13. The li_info Data Structure.

first field of this structure, dpi_curr_db. is itself a record and contains currency information on the database being accessed by a user. The second field, dpi_file, is also a record. The file record contains the file descriptor and file identifier of a file of Daplex transactions, either requests or database descriptions. The next field, dpi_dml_tran. is also a record, and holds information that describes the Daplex transactions to be processed. This includes the number of requests to be processed, the first request to be processed, and the current request being processed. The fourth field of the dap_info record, dap_operation. is a flag that indicates the operation to be performed. This may be either the loading of a new database, or the execution of a request against an existing database. The next

```
struct  dap_info
   {
     struct   curr_db_info        dpi_curr_db;
     struct   file_info           dpi_file;
     struct   tran_info           dpi_dml_tran;
     int      dap_operation;
     struct   ddl_info            *dpi_ddl_files;
     union    kms_info            dpi_kms_data;
     union    kfs_info            dpi_kfs_data;
     union    kc_info             dpi_kc_data;
     int      dap_error;
     int      dap_answer;
     int      dap_buff_count;
   };
```

Figure 3.14. The dap_info Data Structure.

field, dpi_ddl_files, is a pointer to a record describing the descriptor and template files. These files contain information about the ABDL schema corresponding to the current functional database being processed, i.e., the ABDL schema information for a newly defined functional database. The next three fields, dpi_kms_data, dpi_kfs_data and dpi_kc_data, are unions that contain information required by KMS, KFS and KC, respectively. The next field, error, is an integer value representing a specific error type. The next field, answer, is used by LIL to record answers received through its interaction with the user of the interface. The last field, buff_count, is a counter variable used in KC to keep track of the result buffers.

IV. THE LANGUAGE INTERFACE LAYER

The function of the language interface layer (LIL) module is to control the order in which the other modules are called, and allow the user to input transactions from either a file or the terminal. A transaction may take the form of either a database description (DBD) of a new database, or a Daplex request against an existing database. A single transaction may contain multiple requests, allowing a group of requests to perform a single task. For example, several "atomic" statements, those statements that are executed as an indivisible action with respect to the database, could be executed together as a single transaction.

The mapping process occurs when LIL sends a single transaction to KMS. After the transaction has been received by KMS, KC is called to process the transaction. Control always returns to LIL, where the user may either continue with another transaction or close the session by exiting to the operating system.

LIL is menu-driven, and when the transactions are read from either a file or the terminal, they are stored in the dap_req_info data structure. If the transactions are database descriptions, they are sent to KMS in sequential order. If the transactions are Daplex requests, the user is prompted by another menu to selectively choose an individual request to be processed. The menus provide an easy and efficient way for the user to view and select the methods of request

processing desired. Each menu is tied to its predecessor, so that by exiting one menu the user is moved up the "menu tree" This allows the user to perform multiple tasks in one session.

A. THE LIL DATA STRUCTURES

LIL uses three data structures to store the user's transactions and control the transaction sent to KMS. It is important to note that these data structures are shared by both LIL and KMS.

The first data structure is named tran_info and is shown in Figure 4.1. The first field of this record, ti_first_req, is the pointer to the first request data structure that contains the union of all the language requests of MLDS (see Figure 4.2). The first request can originate from either a file or a terminal. The second field of tran_info is a pointer to the current transaction, set by LIL to tell KMS the precise transaction to process next. The third field contains the number of transactions currently in the transaction list. This number is used for loop

```
struct  tran_info
   {
     union    req_info     ti_first_req;
     union    req_info     ti_curr_req;
     int      ti_no_req;
   };
```

Figure 4.1. The tran_info Data Structure.

```
union  req_info
   {
    struct   rel_req_info   *ri_rel_req;
    struct   hie_req_info   *ri_hie_req;
    struct   net_req_info   *ri_net_req;
    struct   dap_req_info   *ri_dap_req;
    struct   ab_req_info    *ri_ab_req;
   };
```

Figure 4.2. The req_info Data Structure.

control when printing the transaction list to the screen. or when searching the list for a transaction to be executed.

The second data structure used by LIL. req_info, is a union of the language requests of MLDS, and is shown in Figure 4.2. It serves a routing control function, in that it routes a transaction request to the appropriate database language. In this thesis, we are concerned only with the the fourth field of this structure, which contains a pointer to the dap_req_info data structure (see Figure 4.3), each copy representing a Daplex user transaction.

The third data structure used by LIL is named dap_req_info. Each copy of this record represents a user transaction, and thus, is an element of the transaction list. The dap_req_info data structure is shown in Figure 4.3. The first field of this record, dap_req, is a character string that contains the actual Daplex transaction. The second field, dap_req_len, contains the length of the transaction. It is used to allocate the exact, and therefore minimal, amount of

```
struct dap_req_info
   {
    char      *dap_req;
    int       dap_req_len;
    struct    temp_str_info      *dap_in_req;
    struct    dap_req_info       *dap_sub_req;
    struct    dap_req_info       *dap_next_req;
   };
```

Figure 4.3. The dap_req_info Data Structure.

memory space for the transaction. The third field, dap_in_req, is a pointer to a list of character arrays that each contains a single line of one transaction. After all lines of a transaction have been read, the line list is concatenated to form the actual transaction, dap_req. If a transaction contains multiple requests, the fourth field, dap_sub_req, points to the list of requests that make up the transaction. In this case, the field dap_in_req is the first request of the transaction. The last field, dap_next_req, is a pointer to the next transaction in the list of transactions.

B. FUNCTIONS AND PROCEDURES

LIL makes use of a number of functions and procedures in order to create the transaction list, pass elements of the list to KMS, and maintain the database schemas. These functions and procedures had been implemented by Anthony and Billings [Ref. 11]. We have made some changes and additions to the original

implementation. As the main structure of the LIL module remains basically unchanged, we do not repeat the description of the LIL processes here. Instead, we document the changes and additions in this section. The updated C code implementation of the LIL module is listed in Appendix C.

1. Changes to the Original Implementation

As mentioned in Chapter I, there are four similar sets of software modules, LIL, KMS, KC, and KFS, required for the each of the four different data language interfaces. Due to the similarity between these interfaces, it is not surprising that the corresponding modules in different interfaces are very similar. The detailed codes of the corresponding modules may not be identical, but the functionalities are mostly the same. As a result, identical procedure names were used for the similar procedures in different implementations of the interfaces. For example, every LIL module in the interfaces provides a procedure that allows the user to load in a new database and all implementations used *load_new* as the procedure name. This posts no problems when the interfaces are compiled and run individually. However, if the four interfaces are to be integrated, it is impossible to link four different procedures having the same name, *load_new*. There are quite a number of these similar procedures in LIL. To avoid this problem for later integration of MLDS, we prefixed the procedure names in the Daplex interface by "f_" which stands for functional. The procedure names that were changed are f_language_interface_layer, f_load_new, f_process_old, f_kernel_controller, f_kernel_mapping_system, f_dbd_to_KMS, f_free_request,

f_rd_temp_str_info. f_read_transaction_file. f_read_terminal, and f_read_file. The C code source files that were affected are main.c, lil.c, lilcommon.c, and readrtnes.c (see Appendix C).

While going through the program codes to check for procedures with identical names, we found three procedures not only with identical name but identical functionality also. These procedures are get_new_user, get_ans, and to_caps. In this case, there is no necessity to duplicate the procedures of identical function with different names in different interfaces. There are extracted from LIL and included in the common routine file, mdbsgenerals.c. As get_ans was the only procedure in the source file getans.c, and get_new_user was the only procedure in the source file getuser.c, these two files were deleted. The procedure to_caps was removed from the source file lilcommon.c.

2. Additional Procedures

For the interface between MLDS and MBDS, template files and descriptor files are required. These files are to be created in the LIL module of MLDS. Procedures for creation of template and description files have been implemented for all of the interfaces except the Daplex interface [Ref. 23]. In this section, we describe the implementations of these procedures for the Daplex interface.

a. Creating the Template File

A *template* is a specification of record structure that the database administrator uses to characterize the organization of records in a file for an

attribute-based database. A record is defined to be a collection of attributes. We can describe the structure of a record in terms of the number of attributes, the names of the attributes, and the associated data types and values. In doing so, we can separate the description of the record away from the actual records and keep the record description in a template. The template can later be used for determining and specifying the characteristics of an attribute and its relation with other attributes in a record. When the records are collected to form a file, the file structure would have the same attributes and similar relations among records in the same file. Because the structural information is maintained in a single template, a file structure can be reorganized by simply changing the template.

The template files in the interfaces have a specific structure. The format of a template file for a database with n files, hence n templates, is shown in Figure 4.4. A typical template description for a record with m attributes is

---

Database-name
Number-of-templates
Template-description-1
Template-description-2
.
.
.
Template-description-n

Figure 4.4. The Template File Format.

---

given in Figure 4.5. The first field gives the number of attributes in the template. Note that this number is always one more than the number of attributes in the record, i.e., m + 1. This is because a constant attribute, FILE, is always added before the actual attributes of the record. The data type in the template description is a single character field which can be s, i or f representing string, integer or float type.

The template file for the Daplex interface is created by transforming the Daplex data structure into the template file structure. First, the data structure fun_dbid_node in Figure 3.2 is read to get the database name and the number of templates in the database. The number of templates is obtained by adding the number of entity type nodes, fdn_num_ent, and the number of generalized subnodes, fdn_num_gen, in the database. Subsequently, the two linked list structures, ent_node in Figure 3.3 and gen_sub_node in Figure 3.4, are

---

```
Number-of-attributes
Template-name
FILE          s
attribute-1   data-type-1
attribute-2   data-type-2
      .
      .
      .
attribute-m   data-type-m
```

Figure 4.5. A Typical Template Description.

---

traversed to extract the required information for each entity type node and generalized subnode. Figure 4.6 represents an algorithm for the transformation. The C code implementation is listed in Appendix B.

Assertions:

1. The Daplex database D has m entity-type nodes $\{E_1, E_2, \ldots, E_m\}$.
2. The Daplex database D has n generalized subnodes $\{S_1, S_2, \ldots, S_n\}$.
3. Each entity-type node E_i, i = 1, . . . , m, has the entity-type name E_i-name.
4. Each generalized subnode S_i, i = 1, . . . , n, has the entity-type name S_i-name.
5. Each E_i, i = 1, . . . , m, has A_{E_i} attributes.
6. Each S_i, i = 1, . . . , n, has A_{S_i} attributes.
7. Each attribute $A_{i,j}$, j = 1, . . . , A_{E_i} or A_{S_i} has the attribute name $A_{i,j}$-name.
8. Each attribute $A_{i,j}$, j = 1, . . . , A_{E_i} or A_{S_i} has the attribute type $A_{i,j}$-type.

Algorithm:

```
write Database-name
write Number-of-templates
/* Repeat for each entity-type node in database */
for each entity-type node E_i in database D do
    {
    write (A_Ei + 1)                    /* Number of attributes */
    write E_i-name                      /* Entity name */
    write "FILE s"
    /* Repeat for each attribute in the entity-type node */
    for each attribute A_i,j in entity-type node E_i do
        {
        write A_i,j-name   A_i,j-type   /* Attribute name, type */
        }
    }
/* Repeat for each generalized subnode in database */
for each generalized subnode S_i in database D do
    {
    write (A_Si + 1)                    /* Number of attributes */
    write S_i-name                      /* Entity subtype name */
    write "FILE s"
    /* Repeat for each attribute in the generalized subnode */
    for each attribute A_i,j in generalized subnode S_i do
        {
        write A_i,j-name   A_i,j-type   /* Attribute name, type */
        }
    }
```

Figure 4.6. Algorithm for Creating the Template File.

---

b. Creating the Descriptor File

While the template file is used to define the record structures of the database, the descriptor file is used to reflect the semantic meanings and intended use of the data. In the descriptor file, the user specifies the attributes (or fields) to be regarded as "key" or "indexing" attributes (fields). MBDS, i.e., KDS, utilizes this information to create the index (cluster) arrangements that permit the most rapid and efficient response to transactions sent to the database system.

The descriptor files in the interfaces also has a specific structure. The format of a descriptor file that has n descriptors is shown in Figure 4.7. The first

---

```
Database-name
FILE B
! Name-of-first-entity
! Name-of-second-entity
                .
                .
                .
! Name-of-last-entity
@
Descriptor-definition-1
Descriptor-definition-2
                .
                .
                .
Descriptor-definition-n
$
```

Figure 4.7. The Descriptor File Format.

---

entry in the format gives the name of the database. The "FILE B" on the second line is a constant that must always be there. Subsequently, for each entity (entity type or subtype) in the Daplex database, a line is added beginning with an exclamation mark "!" and a blank space. followed by the entity-type name. At the end of this list, an at-sign "@" is added to indicate the end of the basic set of descriptors for a given database. It is then followed by a sequence of descriptor definitions. The $ sign at the last line of the format indicates the end of the descriptor file. Each descriptor definition in the descriptor file is expressed in terms of the function. its associated descriptor type and data type. and followed by the Range-or-Equality-Statements as shown in Figure 4.8. The descriptor type can be either A or B which reflects the user's choice of designating the indexes for a function as either a range (A) or an equality (B). The data type can be i, s or f, representing integer, string or float type. The Range-or-Equality-Statements represents a sequence of statements reflecting the selected range or equality values depending on the descriptor type specified. If the descriptor type is A, then the

```
Attribute    Descriptor-type    Data-type
{
  Range-or-Equality-Statements
}
@
```

Figure 4.8. The Typical Descriptor Definition.

sequence of statements are just lines of two numbers specifying the lower limit and upper limit of the selected range. An example of a descriptor definition with range type statements is shown in Figure 4.9. Note that although more than one range can be specified. the ranges must be mutually exclusive.

```
salary      A       f
1000.00     2000.00
3000.00     3500.00
5000.00     9000.00
@
```

Figure 4.9. An Example of a Range Type Descriptor.

Figure 4.10 gives an example of an equality type descriptor. An equality type statement starts with an exclamation mark followed by an exact function value. In our example, the descriptor is for establishing equality index terms for selection of the function *dept* to include the departments specified.

```
dept    B    s
! Computer science
! Electrical engineering
! Mathematics
! Weapon systems
@
```

Figure 4.10. An Example of an Equality Type Descriptor.

Unlike the creation of the template file which is accomplished without the user's knowledge or participation, the creation of the descriptor file involves the user. The user must respond to a sequence of prompts in order to determine the set of descriptors to include in the descriptor file. It is, in fact, the user's detailed knowledge of the database and intended uses of the database that will permit an effective choice of the descriptors.

The algorithm for creating the descriptors of a Daplex database is given in Figure 4.11. This algorithm is basically similar to the those for the other three interfaces [Ref. 23]. The main difference is that there are two data structures, ent_node and gen_sub_node, to be traversed instead of one as in the other interfaces. Also, an important point to note is that Daplex allows a function in an entity which is of entity type. In this case, the user is not permitted to specify descriptors for this function.

Assertions:
1. The Daplex database D has m entity-type nodes $\{E_1, E_2, \ldots, E_m\}$.
2. The Daplex database D has n generalized subnodes $\{S_1, S_2, \ldots, S_n\}$.
3. Each entity-type node E_i, i = 1, . . . , m, has the entity-type name E_i-name.
4. Each generalized subnode S_i, i = 1. . . . , n, has the entity-type name S_i-name.
5. Each E_i, i = 1, . . . , m, has A_{E_i} attributes.
6. Each S_i, i = 1, . . . , n, has A_{S_i} attributes.
7. Each attribute $A_{i,j}$, j = 1, ... A_{E_i} or A_{S_i} has the attribute name $A_{i,j}$-name.

Figure 4.11. Algorithm for Creating the Descriptor File. (continued)

```
Algorithm:
  write Database-name
  write "FILE  B"
  /* Repeat for each entity-type node in database */
  for each entity-type node E_i in database D do
       write  "!"  E_i-name
  /* Repeat for each entity subnode in database */
  for each entity subnode S_i in database D do
       write  "!"  S_i-name
  write  "@"
  /* Repeat for each entity-type node in database */
  for each entity-type node E_i in database D do
       {
       /* Repeat for each attribute in entity-type node */
       for each attribute A_i,j in entity-type node E_i do
            {
            if (A_i,j is not of entity type) then
                 if (A_i,j already selected as index term) then
                      if (more values to be added) then
                           write additional indexing values
                 else
                      if (A_i,j is to be index term) then
                           write  A_i,j   A|B
                           write indexing values
                           write  "@"
            }
       }
  /* Repeat for each entity subnode in database */
  for each entity subnode S_i in database D do
       {
       /* Repeat for each attribute in entity subnode */
       for each attribute A_i,j in entity-type node S_i do
            {
            if (A_i,j is not of entity type) then
                 if (A_i,j already selected as index term) then
                      if (more values to be added) then
                           write additional indexing values
                 else
                      if (A_i,j is to be index term) then
                           write  A_i,j   A|B
                           write indexing values
                           write  "@"
            }
       }
  write  "$"
```

Figure 4.11. Algorithm for Creating the Descriptor File.

V. THE KERNEL MAPPING SYSTEM

The kernel mapping system (KMS) module is called from the language interface layer (LIL) when LIL has received Daplex requests input by the user. The function of KMS is to: (1) parse the request to validate the user's Daplex syntax, and (2) translate, or map, the request to an equivalent ABDL request.

The remainder of this chapter is organized into four parts. In the first part, we present an overview of the general mapping process, focusing on the system structure and the design philosophy. In the second part, we describe how the data definition portion of Daplex is mapped. In the third part, we describe how the data manipulation portion of Daplex is mapped. Finally, in the fourth part, we outline the outstanding work to be done to complete the implementation.

A. AN OVERVIEW OF THE MAPPING PROCESS

From the description of the KMS functions above we immediately see the requirement for a parser as a part of the KMS. This parser validates the Daplex syntax of the input request. The parser is also responsible for doing the translation of Daplex to ABDL. Thus, the parser is the driving force behind the entire mapping system.

1. The KMS Parser / Translator

The KMS parser has been constructed by utilizing Yet-Another-Compiler Compiler (YACC) [Ref. 23]. YACC is a program generator designed for syntactic processing of token input streams. Given a specification of the input language structure (a set of grammar rules), and the user's code. YACC generates a program that syntactically recognizes the input language and allows invocation of the user's code throughout this recognition process. The class of specifications accepted is a very general one: LALR(1) grammars. It is important to note that the user's code mentioned above is our mapping code that is going to perform the Daplex-to-ABDL translation. We utilize a Lexical Analyzer Generator (LEX) [Ref. 24]. LEX is a program generator designed for lexical processing of character input streams. The major difference between LEX and YACC is the specification format for the rules. In the LEX specification, the rules are regular expressions similar to those used in many text editors. Given a regular expression specification of the input strings, LEX generates a program that partitions the input stream into tokens and communicates these tokens to the parser.

The parser produced by YACC consists of a finite-state automaton with a stack that performs a top-down parse, with left-to-right scan and a one token look-ahead. Control of the parser begins initially with the highest-level grammar rule. Control descends through the grammar hierarchy, calling lower and lower-level grammar rules while searching for appropriate tokens in the input. As

the appropriate tokens are recognized, some portions of the mapping code are invoked directly. In other cases, tokens are propagated upwards through the grammar hierarchy until a higher-level rule has been satisfied, and a further translation is performed. When all of the necessary lower-level grammar rules have been satisfied and control has ascended to the highest-level rule, the parsing and translation processes are complete. In Section B and C, we give illustrative examples of these processes. We also describe the subsequent semantic analysis necessary to complete the mapping process.

2. The Grammar Rules

To begin with the implementation specification for KMS, we utilized the Adaplex grammar in the Backus-Naur Form (BNF), provided to us by Dr. Stephen Fox of CCA [Ref. 25]. The Adaplex BNF grammar was then reduced to a pure Daplex BNF grammar by eliminating all system dependent and Ada related goals and rules [Ref. 11]. In this thesis, we further modified the grammar for three reasons. First, the original grammar was too complicated for implementation. There were many layers of recursive rules that made the grammar difficult to comprehend and implement. We modified the grammar in a way that recursions in the grammar were reduced to the minimum without losing the flavor of the Daplex language. Second, when YACC was used to implement the grammar interleaved with C procedures, ambiguities for some rules with multiple alternatives of similar formats often occurred. In most cases, YACC would invoke its disambiguating rules [Ref. 23: pp. 11-14] and produce a correct

parser by selecting one of the valid steps wherever it has a choice. However, if a procedure (action) must be done before the parser can be sure which rule is being recognized, then the application of disambiguating rules is inappropriate. In this case, YACC issues an error message and ignore all conflicting rules except the first one. Third, this and prior language interfaces for SQL, DL/I, and CODASYL-DML have stressed implementations of a subset of each language. This rich subset provides all of the basic, primary database operations. Our implementation for Daplex follow this philosophy.

We now would like to demonstrate the problem of ambiguity by an example. In Figure 5.1, The grammar rule for <set-constructor>, which has four alternatives, is perfectly acceptable in BNF form. To implement this grammar rule in YACC, we would have codes that are in the form shown in Figure 5.2. In this example, there is ambiguity on whether procedure-2 or procedure-3 should be used when the parser sees a LCB followed by a simple_expr. The resulting parser created by YACC then has to choose to recognize only the first, second and fourth

---

<set-constructor> ::=
    LCB <basic-expr-list> RCB
  | LCB <simple-expr> IN <name1> WHERE <dap-expr> RCB
  | LCB <simple-expr> IN <dap-range> WHERE <dap-expr> RCB
  | LCB RCB

Figure 5.1. The Grammar Rule For <set-constructor>.

```
set_constructor :
      LCB { procedure-1 } basic_expr_list { procedure-1a } RCB
    | LCB { procedure-2 } simple_expr IN name1
            WHERE{ procedure-2a }  dap_expr RCB
    | LCB { procedure-3 } simple_expr IN dap_range
            WHERE { procedure-3a } dap_expr RCB
    | LCB { procedure-4 } RCB
```

Figure 5.2. The Implementation of <set-constructor> in YACC.

alternative rules. Any statement written in the third form is reported as having syntax error by the parser. Therefore, it is necessary to rewrite the grammar rules so that the same inputs are read but there are no conflicts. The modified version of the grammar in our implementation is listed in Appendix D.

B. MAPPING THE DATABASE DEFINITION LANGUAGE

In this section, we describe the additional data structures required to implement the database definitions portion of the KMS. This is followed by the a general description of the mapping algorithms.

1. The Data Structures

In addition to the data structures described in Chapter III, KMS also requires some data structures for use during the mapping process. We describe two of these data structures that are shared by the database definitions portion and the database manipulation portion here. Those data structures that are used solely in the manipulation portion are discussed in a later section. At the end of

the mapping process, and before control is returned to LIL, all of the data structures unique to KMS that have been allocated during the mapping process are freed.

The first data structure is dap_kms_info, shown in Figure 5.3. It is used to accumulate information during the grammar parse. It is a record that allows the information to be saved until a point in the parsing processing where it may be utilized in the appropriate portion of the translation process. The first three fields in this data structure, point to the same data structure, ident_list, which

```
struct dap_kms_info
  {
   struct    ident_list          *dki_temp_ptr;
   struct    ident_list          *dki_name1_ptr;
   struct    ident_list          *dki_id_ptr;
   struct    sub_node_list       *dki_overfirst_ptr;
   struct    der_non_node        dki_der_non;
   struct    sub_non_node        dki_sub_non;
   struct    ent_non_node        dki_ent_non;
   struct    function_node       dki_funct;
   struct    ent_value           *dki_ev_ptr;
   struct    dap_create_list     *dki_create;
   struct    req_line_list       *dki_req_ptr;
   struct    create_ent_list     *dki_cel_ptr;
   struct    overlap_node        *dki_create_ovrptr;
   struct    ent_value_list      dki_evl_ptr;
   struct    dml_statement       *dml_statement_ptr;
   struct    loop_info           *loop_info_ptr;
  };
```

Figure 5.3. The dap_kms_info Data Structure.

temporarily holds a list of names for comparison with the identifiers. Whenever a list of identifiers is read, the name of each identifier is kept in the linked list pointed to by dki_temp_ptr. The dki_temp_ptr always points to a list that contains the names of the identifiers in the current list. When there are more than one list of identifiers to be compared, for example in the overlapping rule declaration, both the dki_name1_ptr and dki_temp_ptr pointer are used. For type or subtype declarations, both the dki_temp_ptr and the dki_id_ptr are used. The dki_id_ptr points to a linked list that contains the names of all type and subtype identifiers declared so far. Whenever a new type or subtype declaration is parsed, the name of the new type or subtype identifier is checked against this list to ensure no duplicate declarations. The dki_overfirst_ptr is used when parsing an overlapping rule declaration to keep the pointers to the gen_sub_node in the first list of the declaration. The next four fields are used to store information for der_non_node, sub_non_node, ent_non_node and function_node temporarily. The dki_ev_ptr holds a pointer to a temporary storage of entity values. The last six fields hold temporary storage or pointers to data structures used solely by the database manipulation portion (see Section C).

The ident_list data structure, depicted in Figure 5.4, contains only two fields. The first field is a character array that holds the identifier name. The second field points to the next ident_list component. This data structure provides a linked list to maintain identifier names.

```
struct ident_list
   {
    char      name [ENLength + 1];
    struct    ident_list      *next;
   };
```

Figure 5.4. The ident_list Data Structure.

2. The Algorithm

The mapping process of the database definitions portion in KMS is relatively straightforward. It involves the parsing of the database schema and the transformation of the corresponding database definitions onto the data structures described in Chapter III. The detailed implementation of the mapping using YACC and C is given in Appendix C. In this section, we provide a general description of the algorithm. The schema of the sample database, *univ*, (see Figure 2.1 again) represents a typical database declaration in Daplex. It includes the type and subtype declarations, the entity and non-entity declarations, the function declarations, the overlapping constraint declarations, and the uniqueness constraint declarations. We test our implementation of the DDL portion of KMS by parsing this database schema.

For each database declaration, the database name is checked against the current list of databases in the system which is the linked list pointed to by dn_fun in dbid_node (see Figure 3.1). If the database name has been declared

before, the transaction is rejected. Otherwise, a new ent_dbid_node is allocated and appended to the current list of databases.

For type declarations and subtype declarations, similar checks are made to ensure that there is no duplication of type or subtype names. These declarations include those for the entity type, the entity subtype, the nonentity base-type, the nonentity subtype and the derived nonentity type. KMS recognizes the difference between these declarations by their different formats and creates corresponding data structures to store the information. The data structures are ent_node (Figure 3.3), gen_sub_node (Figure 3.4), ent_non_node (Figure 3.5), sub_non_node (Figure 3.6) and der_node_node (Figure 3.7). When there is no error detected, an appropriate node is created and appended to the corresponding linked list pointed to in the data structure ent_dbid_node. The corresponding counter in ent_dbid_node is also updated.

For entity type or subtype declarations, incomplete specification is allowed. In this case, a skeleton structure of the ent_node or gen_sub_node with only the name field filled is created. When the complete declarations is found later, then the other necessary information are inserted into the structure.

For function declarations, the function_node data structure (Figure 3.9) is used. Depending on the function type declared, the appropriate fields in the function_node structure are updated. When more than one function are declared, a linked list of function_node is created. At the end of the function declarations, the linked list that has been created is then added to the ent_node or

gen_sub_node under which the functions are declared. If the function type is entity, then the ent_node or gen_sub_node of this entity is recalled and its indicator in the data structure, en_terminal or gsn_terminal, is updated to reflect that this entity is not a terminal.

For overlapping constraint declarations, two identifier lists are read in. Error checkings are performed to ensure that the identifiers in the lists are terminal generalized subnode types and they all have the same base type. The overlapping information are then stored in the data structure overlap_node (Figure 3.8) and appended to the edn_ovrptr of the ent_dbid_node.

For the uniqueness constraint rule, we traverse the ent_node and the gen_sub_node data structures to search for the entity-type node or generalized subnode that matches the entity identifier declared. Subsequently, the function_node data structures are traversed to locate the function nodes and the fn_unique field of each of the function nodes found is set to true.

Three identifiers list rules are used in the implementation. They are new_id_list, id_list and name1_list. The basic intent of the three rules are the same - to recognize a sequence of identifiers. However, three different rules are used because different actions are needed in various situations. The new_id_list rule detects duplicate declarations of name_id against dki_id_ptr but does not issue an error message. Instead, an error flag is turned on so that the procedure immediately following the rule has the option of ignoring duplication. This is necessary because incomplete declarations of entity types are allowed in Daplex.

The id_list rule checks against dki_temp_ptr and displays error messages if there is a duplication. The name1_list rule checks duplication against dki_name1_ptr and issues an error message when required.

C. MAPPING THE DATA MANIPULATION LANGUAGE

In this section, we describe the data structures that are used solely in the DML portion of KMS. This is followed by a general description of the mapping algorithm.

1. The Data Structures

The database manipulation portion of KMS uses many data structures for temporary storage of information. They are dap_create_list, dap_av_pair_list, req_line_list, create_ent_list, dap_expr_info, relation_list, simple_expr1, simple_expr2, simple_expr3, simple_expr4, funct_appln, indexed_component, set_constructor, set_construct2, set_construct3, dap_range_info, basic_expr_list, comp_assoc_list, domain_info, loop_info, order_comp_list, dml_statement2_list, dml_statement. These data structures, depicted in Appendix A, are directly derivable from the grammar rules. For example, the data structure domain_info is derivable from the domain rule in the grammar.

Why are we taking this approach? When we first started on the implementation of the database manipulation portion of KMS, we tried to translate the Daplex transactions directly to the ABDL transactions. However, we encountered some problems and could not proceed in this approach due to the

recursive nature of the Daplex grammar rules. Another approach to the problem was needed. So, we built the data structures described above to correspond to the database manipulation portion of the grammar rules. During the parsing process, information on the transactions, conforming to the Daplex grammar rules, are transferred to the corresponding data structures. At the end of the parsing process, separate procedures would then be used to read from the data structures and convert them into ABDL transaction formats.

2. The Algorithm

When the user wishes LIL to process Daplex requests against an existing database, the task of KMS is to translate the user's Daplex request to equivalent ABDL requests. The Daplex request must also be specified within a rigid set of grammar rules (see Appendix D).

The DML statements in the Daplex grammar are represented by the dml_statement rule. This rule includes statements for CREATE, DESTROY, MOVE and FOR EACH transactions. These statements, by themselves, can be executed as a valid transaction request. The second rule, dml_statement2, handles those transactions that can be executed within a loop (FOR EACH) statement. The statements that can only be included within a loop are the assignment-statement, the include-statement, the exclude-statement, the destroy-statement, the move-statement and the procedure-call statement.

As mentioned in the previous section, the first step in the translation of a Daplex transaction to the ABDL request(s) is to instantiate the data structures

corresponding to the grammar rules parsed. We chose the names of the data structures as close as possible to the corresponding grammar rules. For example, dap_expr_info for the dap_expr rule and set_constructor for the set_constructor rule.

Rules that have more than one alternatives are handled in two different ways, depending on the situation. When each of the alternative rules has a single corresponding data structure, then the new data structure is just a collection of pointers to those corresponding data structures. An example of this is the simple_expr1 data structure for the simple-expr rule (Figure 5.5). At any one time, only one field in the simple_expr1 data structure is active. In the case that

---

```
<simple-expr> ::=
        <literal>
    |   <set-constructor>
    |   <indexed-component>
    |   <function-application>

struct simple_expr1
   {
    char      lit_array[LITLength+1];
    struct    set_constructor          *set_construct_ptr;
    struct    indexed_component        *indexed_comp_ptr;
    struct    funct_appln              *funct_appln_ptr;
   };
```

Figure 5.5. The Grammer Rule and Data Structure for <simple-expr>.

---

the alternative rules do not correspond to single data structures, new data structures are created for the alternative statements. For example, the relation rule has four alternative statements. The corresponding data structure is the relation_list shown in Figure 5.6. Four pointers to the data structures, simple_expr1, simple_expr2, simple_expr3 and simple_expr4, are utilized and correspond to the four alternative statements.

```
<relation> ::=
        <simple-expr>
      | <simple-expr> <relational-operator> <simple-expr>
      | <simple-expr> <in-op> <dap-range>
      | <simple-expr> <in-op> <name1>

struct relation_list
   {
    struct    simple_expr1       *simple_expr1_ptr;
    struct    simple_expr2       *simple_expr2_ptr;
    struct    simple_expr3       *simple_expr3_ptr;
    struct    simple_expr4       *simple_expr4_ptr;
    struct    relation_list      *next;
   };
```

Figure 5.6. The <relation> Rule and The relation_list Data Structure.

D. FUTURE WORKS

As we reached this stage in the implementation, a decision was made to discontinue the implementation effort for this thesis and leave the remainder for

future efforts due to a time limitation. In this section, we outline the work that needs to be done to complete the implementation.

1. Completing the Kernel Mapping System

The DDL portion of KMS has been fully implemented. As for the DML portion, the implementation of the parser that translates the Daplex database manipulation request to the corresponding data structures has been completed. However, the procedures for mapping the data structures to the ABDL request have not been implemented.

To complete the implementation of KMS, we need a procedure to access the dml_statement_ptr field of the dap_kms_info data structure (see Figure 5.3). The dml_statement_ptr points to the data structure dml_statement, as shown in Figure 5.7. There are four pointers in the dml_statement data structure which point to the dap_expr_info, the indexed_component, the basic_expr_list and the

```
struct dml_statement
   {
    int       type;
    struct    dap_expr_info              *dap_expr_ptr;
    struct    indexed_component *indexed_comp_ptr;
    struct    basic_expr_list            *basic_expr_ptr;
    struct    comp_assoc_list            *comp_assoc_ptr;
   };
```

Figure 5.7. The dml_statement Data Structure.

comp_assoc_list data structures. Four additional procedures, one for each of the aforementioned data structures, are required to read from the data structures and translate the information in the data structures into ABDL request(s).

2. The Kernel Controller

Once KMS has performed the necessary transformations and translations of Daplex requests into their equivalent ABDL requests. KC controls the submission of the ABDL requests to MBDS for processing. After ABDL processed the requests, control is returned to LIL.

Just as data structures are specified for LIL and KMS, we also require data structures that are specified for KC to recognize the different types of ABDL requests. We need a KC procedure to control the processing of each of the ABDL request (i.e., INSERT, DELETE, UPDATE, and RETRIEVE).

3. The Kernel Formatting System

KFS is the fourth module in the language interface, and is called by the kernel controller when it is necessary to display results to the user. The only task that KFS performs is to display on the screen the result returned by MBDS in Daplex format. Therefore, it is the easiest module to implement in the language interface.

Both the kernel controller and the kernel formatting system have not been implemented yet. However, in our implementation of LIL and KMS, a stub is inserted wherever KC or KFS is to be called. The name of the stubs are f_kernel_controller and f_kernel_formatting_system, respectively. So, the next

implementor may proceed to implement the two modules using the same names. After the two modules are implemented, all we need to do for integrating the four modules, i.e., LIL, KMS, KC, and KFS, is to replace the stubs.

VI. THE CONCLUSION

In this thesis, we have presented a partial specification and implementation of a Daplex language interface . This is one of four language interfaces that the multi-lingual database system supports. When complete, the multi-lingual database system will be able to execute transactions written in four well-known and important data languages, namely, SQL, DL/I, Daplex, and CODASYL. The work accomplished in this thesis is part of the ongoing research effort being conducted at the Laboratory for Database Systems Research, located at the Naval Postgraduate School in Monterey, California.

The need to provide an alternative to the development of separate stand-alone database systems for specific data model/data language constructions has been the motivation for this research. In this regard, we have first demonstrated the feasibility of a multi-lingual database system (MLDS) by showing how a software Daplex language interface can be constructed.

A major goal has been to design a Daplex-to-ABDL interface without requiring any change be made to ABDL or the underlying database system, MBDS. Our partial implementation may be completely resident on a host computer or the controller. All of the Daplex transactions are executed and processed in the Daplex interface. MBDS continues to receive and process

transactions written in the unaltered syntax of ABDL. In addition, our implementation has not required any change to the syntax of Daplex. The interface is completely transparent to the Daplex user as well as to the MBDS user.

In retrospect, our level-by-level, top-down approach to the design of the interface has been a good choice. This implementation methodology has been the most familiar to us and proved to be relatively efficient in time. In addition, this approach permits follow-on programmers to easily maintain and modify the code.

In this thesis, we have fully implemented the LIL module and partially implemented the KMS module. A minor part of the KMS module and the KC and KFS modules are left for another programmer to follow up. It is a great disappointment that we have not been able to complete the implementation. The primary reason for our failure has been the complexity of the functional model and the Daplex language. The total amount of codes required for the implementation far exceed those required for the other interface implementations. In terms of lines of code, we have implemented the same number of lines as in the entire language interface of the CODASYL implementation. We have also outlined how the remaining implementation should proceed.

We have shown that a Daplex interface can be implemented as part of MLDS. We have provided a partial software structure to facilitate this interface, and we have developed actual code for implementation. The next step is to complete the development of the Daplex interface. When complete, this interface

can be integrated with the other implementations and tested as a whole to determine how efficient, effective, and responsive it can be to a users' needs. The results may be the impetus for a new direction in database system research and development.

APPENDIX A

THE DAPLEX DATA STRUCTURES

```
union  dbid_node
   {
    struct      rel_dbid_node   *dn_rel;
    struct      hie_dbid_node   *dn_hie;
    struct      net_dbid_node   *dn_net;
    struct      fun_dbid_node   *dn_fun;
   }

struct   fun_dbid_node
   /* structure def for each entity-relationship dbid node */
   {
    char        fdn_name[DBNLength + 1];
    struct      ent_non_node    *fdn_nonentptr;
    int         fdn_num_nonent;         /* number of nonentity types */
    struct      ent_node        *fdn_entptr;
    int         fdn_num_ent;            /* number of entity types */
    struct      gen_sub_node    *fdn_subptr;
    int         fdn_num_gen;            /* number of gen_subtypes */
    struct      sub_non_node    *fdn_nonsubptr;
    int         fdn_num_nonsub;         /* number of nonentity subtypes */
    struct      der_non_node    *fdn_nonderptr;
    int         fdn_num_der;            /* number or nonentity derived types */
    struct      overlap_node    *fdn_ovrptr;
    int         fdn_num_ovr;            /* number of overlap_nodes */
    struct      fun_dbid_node   *fdn_next_db;
   };

struct   ent_node
  /* structure definition for each entity node */
   {
    char        en_name[ENLength + 1];
    int         en_last_ent_id;         /* keeps track of the unique id assigned
                                           to each entity type in the database */
    int         en_num_funct;           /* number of assoc. functions  */
    int         en_terminal;            /* if true (=1) it is a terminal type */
    struct      function_node   *en_ftnptr;
    struct      ent_node        *en_next_ent;
   };

struct   gen_sub_node
  /* structure def for each generalization (supertype/subtype) node */
   {
    char        gsn_name[ENLength + 1];
    int         gsn_num_funct;          /* number of assoc. functions*/
    int         gsn_terminal;           /* if true (=1) it is a terminal type */
    struct      ent_node_list   *gsn_entptr;      /* ptr to entity supertype */
    int         gsn_num_ent;            /* number of entity supertypes */
```

```
    struct      function_node  *gsn_ftnptr;
    struct      sub_node_list  *gsn_subptr; /* ptr to subtype supertype */
    int         gsn_num_sub;            /* number of subtype supertypes */
    struct      gen_sub_node   *gsn_next_genptr;
   };

struct  ent_non_node
  /* structure def for each base-type nonentity node */
   {
    char        enn_name[ENLength - 1];
    char        enn_type;               /* either i(nteger), s(tring),
                                           f(loat), e(numeration), or b(oolean) */
    int         enn_total_length;       /* max length of base-type value */
    int         enn_range;              /* true or false depending
                                           on whether there is a
                                           range.  If range exists,
                                           there must be two entries
                                           into ent_value */
    int         enn_num_values;             /* number of actual values */
    struct      ent_value      *enn_value;      /* actual value of base-type */
    int         enn_constant;           /* boolean to refelect constant value */
    struct      ent_non_node   *enn_next_node;
   };

struct  sub_non_node
  /* structure def for each subtype nonentity node */
   {
    char        snn_name[ENLength - 1];
    char        snn_type;               /* either i(nteger), s(tring),
                                           f(loat), e(numeration), or b(oolean) */
    int         snn_total_length;       /* max length of subtype value */
    int         snn_range;              /* true or false depending
                                           on whether there is a
                                           range.  If range exists,
                                           there must be two entries
                                           into ent_value */
    int         snn_num_values;         /* number of actual values */
    struct      ent_value      *snn_value;      /* actual value of subtype */
    struct      sub_non_node   *snn_next_node;
   };

struct  der_non_node
  /* structure def for each derived type nonentity node */
   {
    char        dnn_name[ENLength - 1];
    char        dnn_type;               /* either i(nteger), s(tring),
                                           f(loat), e(numeration), or b(oolean) */
    int         dnn_total_length;       /* max length of derived type value */
    int         dnn_range;              /* true or false depending on whether
                                           there is a range.  If range exists,
                                           there must be two entries in the
```

```
                                                    ent_value */
    int         dnn_num_values;                 /* number of actual values */
    struct      ent_value       *dnn_value;     /* actual value of derived type */
    struct      der_non_node  *dnn_next_node;
   };

struct  overlap_node
  /* structure def for overlapping constraint */
   {
    char        base_type_name[ENLength+1];
    struct      sub_node_list   *snlptr;
    int         num_sub_node; /* number of sub_node in the above list */
    struct      overlap_node    *next;
   };

struct  function_node
  /* structure definition for each function type declaration */
   {
     char       fn_name[ENLength+1];
     char       fn_type:                    /* either f(loat), i(nteger), s(tring).
                                                b(oolean), or e(numeration) */
     int        fn_range:                   /* Boolean if range of values */
     int        fn_set;                     /* Boolean if set of values */
     int        fn_total_length;            /* max length */
     int        fn_num_value;               /* number of actual values */
     struct     ent_value       *fn_value;      /* actual value */
     struct     ent_node        *fn_entptr;     /* ptr to entity type */
     struct     gen_sub_node    *fn_subptr;     /* ptr to entity subtype */
     struct     ent_non_node    *fn_nonentptr;  /* ptr to nonentity type */
     struct     sub_non_node    *fn_nonsubptr;  /* ptr to nonentity subtype */
     struct     der_non_node    *fn_nonderptr;  /* ptr to nonentity dertype */
     int        fn_entnull;                 /* initialized false set true for no value */
     int        fn_unique;                  /* init false - unique if true */
     struct     function_node   *next;
   };

struct  sub_node_list     /* list of ponters */
  /* structure definition for terminal subtypes that define one or more
     subtypes */
   {
    struct      gen_sub_node    *subptr;  /* only terminal subtypes */
    struct      sub_node_list   *next;
   };

struct  ent_node_list     /* list of pointers */
   /* structure definition for subtypes with one or more entity supertypes */
   {
    struct      ent_node        *entptr;
    struct      ent_node_list   *next;
   };
```

```
struct   ent_value
  /* struct      def for value of 'i','s','f','e', or 'b' */
   {
    char        *ev_value;     /* pointer to character string only */
    struct      ent_value      *next;
   };

struct   ident_list
   {
    char        name[ENLength + 1];
    struct      ident_list     *next;
   };

struct   ent_value_list
   {
    char        type;
    int         num_values;
    struct      ent_value      *ev_ptr:
   };

struct   dap_kms_info
   {
    struct      ident_list        *dki_temp_ptr;
    struct      ident_list        *dki_name1_ptr;
    struct      ident_list        *dki_id_ptr;
    struct      sub_node_list     *dki_overfirst_ptr;
    struct      der_non_node      dki_der_non;
    struct      sub_non_node      dki_sub_non;
    struct      ent_non_node      dki_ent_non;
    struct      function_node     dki_funct;
    struct      ent_value         *dki_ev_ptr;
    struct      dap_create_list   *dki_create;
    struct      req_line_list     *dki_req_ptr;
    struct      create_ent_list   *dki_cel_ptr;
    struct      overlap_node      *dki_create_ovrptr;
    struct      ent_value_list    dki_evl_ptr;
    struct      dml_statement     *dml_statement_ptr;
    struct      loop_info         *loop_info_ptr;
   };

struct   dap_create_list
   {
    int         req_type;                 /* Insert or Retrieve */
    char        en_name[ENLength + 1];
    struct      dap_av_pair_list          *av_pair_ptr;
    struct      dap_create_list           *next;
   };


struct   dap_av_pair_list
   {
```

```
    char        name[ENLength + 1];
    struct      function_node           *ftnptr;
    int         num_value;
    struct      ent_value               *valptr;
    struct      dap_av_pair_list        *next;
   };

struct  req_line_list
   {
    char        req_line[REQLength];
    struct      req_line_list           *next;
   };

struct  create_ent_list
   {
    struct      ent_node_list           *enl_ptr;
    struct      sub_node_list           *snl_ptr;
    struct      create_ent_list         *next;
   };

struct  dap_expr_info
   {
    int         relation_type;  /* Relation, AndRelation or OrRelation */
    struct      relation_list           *rel_list_ptr;
   };

struct  relation_list
   {
    struct      simple_expr1            *simple_expr1_ptr;
    struct      simple_expr2            *simple_expr2_ptr;
    struct      simple_expr3            *simple_expr3_ptr;
    struct      simple_expr4            *simple_expr4_ptr;
    struct      relation_list           *next;
   };

struct  simple_expr1
   {
    char        lit_array[LITLength+1];
    struct      set_constructor         *set_construct_ptr;
    struct      indexed_component       *indexed_comp_ptr;
    struct      funct_appln             *funct_appln_ptr;
   };

struct  funct_appln
   {
    int         type;           /* COUNT, SUM, AVG, MIN, MAX */
    /* The followings are structures for expr_types in the grammar */
    char        name_id[ENLength + 1];
    struct      set_constructor         *set_construct_ptr;
    struct      indexed_component       *indexed_comp_ptr;
   };
```

```
struct   indexed_component
   {
    char        name_id[ENLength + 1];
    char        type;      /* Entity, GenSub, LoopParameter, Function */
    char        parent_name[ENLength + 1];
    struct      indexed_component      *next;
   };

struct   set_constructor
   {
    struct      basic_expr_list        *basic_expr_ptr;
    struct      set_construct2         *set_construct2_ptr;
    struct      set_construct3         *set_construct3_ptr;
   };

struct   set_construct2
   {
    struct      simple_expr1           *simple_expr1_ptr;
    char        name1[ENLength + 1];
    struct      dap_expr_info          dap_expr_ptr;
   };

struct   dap_range_info
   {
    int         range_type;      /* Integer or Float */
    char        first_value[FLTLength - 1];
    char        second_value[FLTLength + 1];
   };

struct   set_construct3
   {
    struct      simple_expr1           *simple_expr1_ptr;
    struct      dap_range_info         dap_range;
    struct      dap_expr_info          dap_expr_ptr;
   };

struct   basic_expr_list
   {
    char        lit_array[LITLength+1];
    struct      indexed_component      *indexed_comp_ptr;
    struct      funct_appln            *funct_appln_ptr;
    struct      basic_expr_list        *next;
   };

struct   comp_assoc_list
   {
    char        name[ENLength + 1];
    struct      simple_expr1           simple_expr;
    struct      comp_assoc_list        *next;
   };
```

```
struct  simple_expr2
   {
    struct      simple_expr1            *first_expr;
    int         rel_operator;
    struct      simple_expr1            *second_expr;
   };

struct  simple_expr3
   {
    struct      simple_expr1            *simple_expr;
    int         in_op:          /* INOp or NINOp */
    struct      dap_range_info          *dap_range;
   };

struct  simple_expr4
   {
    struct      simple_expr1            *simple_expr;
    int         in_op:          /* INOp or NINOp */
    char        name_id[ENLength + 1];
   };

struct  domain_info
   {
    /* only one of the first two fields is active at one time */
    /* see loop_expr in grammar */
    struct      indexed_component       *indexed_comp_ptr;
    char        name[ENLength + 1];
    struct      dap_expr_info           *dap_expr_ptr;  /* optional field */
   };

struct  loop_info
   {
    char        loop_parameter[ENLength + 1];
    struct      ent_node                *entptr;
    struct      gen_sub_node            *subptr;
    struct      domain_info             domain;
    struct      order_comp_list         *order_comp_ptr;
    struct      dml_statement2_list     *dml_statement2_list_ptr;
   };

struct  order_comp_list
   {
    int         sort_order;             /* ASCENDING or DESCENDING */
    struct      indexed_component       *indexed_comp_ptr;
    struct      order_comp_list *next;
   };

struct  dml_statement2_list
   {
    struct      dml_statement           *dml_statement2_ptr;
    struct      dml_statement2_list     *next;
```

```
};

struct  dml_statement
  {
   int          type;           /*  Assignment, Include, Exclude,
                                    Destroy, Move, Procedure, Create */
   struct       dap_expr_info          *dap_expr_ptr;
   struct       indexed_component      *indexed_comp_ptr;
   struct       basic_expr_list        *basic_expr_ptr;
   struct       comp_assoc_list        *comp_assoc_ptr;
  };
```

APPENDIX B

THE LIL MODULE

1. FILE : lil.c

```
#include <stdio.h>
#include "licommdata.def"
#include "struct.def"
#include "flags.def"
#include "dap.ext"
#include "lil.dcl"

f_language_interface_layer()
  /*  This proc allows the user to interface with the system.  */
  /*  Input and output: user DAPLEX requests                   */

{
  int    num;
  int    stop;       /* boolean flag */


#ifdef EnExFlag
  printf ("Enter f_language_interface_layer\n");
#endif

 dap_init();

 /* initialize several ptrs to different parts of the user structure */
 /* for ease of access                                               */
 dap_info_ptr  = &(cuser_dap_ptr->ui_li_type.li_dap);
 tran_info_ptr = &(dap_info_ptr->dpi_dml_tran);
 first_req_ptr = &(tran_info_ptr->ti_first_req);
 curr_req_ptr  = &(tran_info_ptr->ti_curr_req);

/* the followings are inserted for testing build_ddl_files only */
/*
 dap_info_ptr->dpi_curr_db.cdi_db.dn_fun = dbs_dap_head_ptr.dn_fun;
 f_build_ddl_files();
*/
/* end test code */

 stop = FALSE;
 while (stop == FALSE)
   {
    /* allow user choice of several processing operations */
    printf ("\nEnter type of operation desired\n");
    printf ("\t(l) - load new database\n");
    printf ("\t(p) - process existing database\n");
    printf ("\t(x) - return to the operating system\n");
```

```
    dap_info_ptr->dap_answer = get_ans(&num);

    switch (dap_info_ptr->dap_answer)
      {
        case 'l': /* user desires to load a new database */
                f_load_new();
                break;
        case 'p': /* user desires to process an existing database */
                f_process_old();
                break;
        case 'x': /* user desires to exit to the operating system */
                /* database list must be saved back to a file   */
                stop = TRUE;
                break:
        default:  /* user did not select a valid choice from the menu */
                printf ("\nError - invalid operation selected\n");
                printf ("Please pick again\n");
                break;
      } /* end switch */

   /* return to main menu */

  } /* end while */

#ifdef EnExFlag
  printf ("Exit f_language_interface_layer\n");
#endif

} /* end f_language_interface_layer */

dap_init()
{

#ifdef EnExFlag
  printf ("Enter dap_init\n");
#endif

#ifdef EnExFlag
  printf ("Exit dap_init\n");
#endif

} /* end dap_init */

f_load_new()
  /*  This proc accomplishes the following:                    */
  /*  (1) determines if the new database name already exists,   */
  /*  (2) adds a new header node to the list of schemas,        */
  /*  (3) determines the user input mode (file/terminal),       */
  /*  (4) reads the user input and forwards it to the parser, and */
  /*  (5) calls the routine that builds the template/descriptor files */
```

```
{
   int   num;
   int   more_input;    /* boolean flag */
   int   proceed;       /* boolean flag */
   struct fun_dbid_node *db_list_ptr,              /* ptr to the current db */
                        *new_ptr,            /* ptr to a new db structure */
                        *fun_dbid_node_alloc(); /* ptr to allocated db */

#ifdef EnExFlag
  printf ("Enter f_load_new\n");
#endif

  /* prompt user for name of new database */
  printf ("[7;7m\nEnter name of database ---->[0;0m ");
  readstr (stdin. dap_info_ptr->dpi_curr_db.cdi_dbname);
  to_caps (dap_info_ptr->dpi_curr_db.cdi_dbname);
  db_list_ptr = dbs_dap_head_ptr.dn_fun;
  while (db_list_ptr != NULL)
   {
    /* determine if new database name already exists */
    /* by traversing list of entity-relation db schemas */
    if ((strcmp(db_list_ptr->fdn_name,
             dap_info_ptr->dpi_curr_db.cdi_dbname))== 0)
     {
      printf ("\nError - db name already exists\n");
      printf ("[7;7mPlease reenter db name ---->[0;0m ");
      readstr (stdin, dap_info_ptr->dpi_curr_db.cdi_dbname);
        to_caps (dap_info_ptr->dpi_curr_db.cdi_dbname);
      db_list_ptr = dbs_dap_head_ptr.dn_fun;
     } /* end if */
    else
      db_list_ptr = db_list_ptr->fdn_next_db;
  } /* end while */

  /* continue - user input a valid 'new' database name */
  /* add new header node to the list of schemas and fill-in db name */
  /* append new header node to db_list & init relevent user stucture ptrs */

  new_ptr = fun_dbid_node_alloc();
  strcpy (new_ptr->fdn_name, dap_info_ptr->dpi_curr_db.cdi_dbname);
  new_ptr->fdn_next_db = dbs_dap_head_ptr.dn_fun;
  dbs_dap_head_ptr.dn_fun = new_ptr;
  dap_info_ptr->dpi_curr_db.cdi_db.dn_fun = new_ptr;
  dap_info_ptr->dpi_curr_db.cdi_attr.an_dattr_ptr = NULL;

  /* check for user's mode of input */
  more_input = TRUE;
  while (more_input == TRUE)
   {
    /* determine user's mode of input */
    printf ("\nEnter mode of input desired\n");
```

```
    printf ("\t(f) - read in database description from a file\n");
    printf ("\t(x) - return to the to main menu\n");
    dap_info_ptr->dap_answer = get_ans(&num);

    switch (dap_info_ptr->dap_answer)
     {
      case 'f': /* user input is from a file */
             f_read_transaction_file();
                  if (dap_info_ptr->dap_error != ErrReadFile)
                   {
                       /* file contains transactions */
                       /* dbd stands for database description */
                    f_dbd_to_KMS();
                    f_free_requests();
                      if (dap_info_ptr->dap_error != ErrCreateDB)
                       {
                         /* no syntax errors in creates */
                   f_build_ddl_files();
                       f_Kernel_Controller();
                        } /* end if */
                   } /* end if */
             break;
      case 'x': /* exit back to LIL */
             more_input = FALSE:
             break;
      default:  /* user did not select a valid choice from the menu */
             printf ("\nError - invalid input mode selected\n");
             printf ("Please pick again\n");
             break;
     } /* end switch */

    if (dap_info_ptr->dap_error == ErrCreateDB)
        /* errors in creates so exit this loop */
        more_input = FALSE;
    dap_info_ptr->dap_error = NOErr;
   } /* end while */


#ifdef EnExFlag
  printf ("Exit f_load_new\n");
#endif

} /* end f_load_new */

f_process_old()
  /*  This proc accomplishes the following:                */
  /*  (1) determines if the database name already exists,   */
  /*  (2) determines the user input mode (file/terminal),   */
  /*  (3) reads the user input and forwards it to the parser */

{
```

```
  int    found, more_input;          /* boolean flags */
  int    num;
  struct fun_dbid_node *db_list_ptr; /* ptr to the current database */


#ifdef EnExFlag
  printf ("Enter f_process_old\n");
#endif

 /* prompt user for name of existing database */
 printf ("[7;7m\nEnter name of database ---->[0;0m ");
 readstr (stdin, dap_info_ptr->dpi_curr_db.cdi_dbname);
 to_caps (dap_info_ptr->dpi_curr_db.cdi_dbname);
 db_list_ptr = dbs_dap_head_ptr.dn_fun;

 found = FALSE;
 while (found == FALSE)
  {
   /* determine if database name does exist      */
   /* by traversing list of entity-relation schemas */
   if (strcmp(dap_info_ptr->dpi_curr_db.cdi_dbname.db_list_ptr->fdn_name)== 0)
       {
    found = TRUE:
    dap_info_ptr->dpi_curr_db.cdi_db.dn_fun = db_list_ptr:
      }
   else
    {
     db_list_ptr = db_list_ptr->fdn_next_db;
     /* error condition causes end of list('NULL') to be reached */
     if (db_list_ptr == NULL)
      {
       printf ("\nError - db name does not exist\n");
       printf ("[7;7mPlease reenter valid db name ---->[0;0m ");
       readstr (stdin, dap_info_ptr->dpi_curr_db.cdi_dbname);
          to_caps (dap_info_ptr->dpi_curr_db.cdi_dbname);
       db_list_ptr = dbs_dap_head_ptr.dn_fun;
      } /* end if */

    } /* end else */

  } /* end while */

 /* continue - user input a valid existing database name */
 /* determine user's mode of input */
 more_input = TRUE;
 while (more_input == TRUE)
  {
   printf ("\n\nEnter mode of input desired\n");
   printf ("\t(f) - read in a group of DAPLEX requests from a file\n");
   printf ("\t(t) - read in DAPLEX requests from the terminal\n");
   printf ("\t(x) - return to the previous menu\n");
```

```
    dap_info_ptr->dap_answer = get_ans(&num);

    switch (dap_info_ptr->dap_answer)
     {
      case 'f': /* user input is from a file */
        f_read_transaction_file();
          dapreqs_to_KMS();
          f_free_requests();
          break;
       case 't': /* user input is from the terminal */
         f_read_terminal();
         dapreqs_to_KMS();
           f_free_requests();
           break;
       case 'x': /* user wishes to return to LIL menu */
         more_input = FALSE;
         break;
       default: /* user did not select a valid choice from the menu */
         printf ("\nError - invalid input mode selected\n");
         printf ("Please pick again\n");
           break;
      } /* end switch */

   } /* end while */

#ifdef EnExFlag
  printf ("Exit f_process_old\n");
#endif

} /* end f_process_old */
```

2. FILE : lilcommon.c

```
#include <stdio.h>
#include <ctype.h>
#include <strings.h>
#include "licommdata.def"
#include "struct.def"
#include "flags.def"
#include "dap.ext"
#include "lil.ext"

f_dbd_to_KMS()
        /* This routine sends the request list of database descriptions one  */
        /* by one to the KERNAL MAPPING SYSTEM and frees the list as it goes */

{

#ifdef EnExFlag
```

```
  printf ("Enter f_dbd_to_KMS\n");
#endif

 dap_info_ptr->dap_operation = CreateDB;

 /* set the current ptr to the first ptr */
 curr_req_ptr->ri_dap_req = first_req_ptr->ri_dap_req;

 /* now send each request to KMS */
 while (curr_req_ptr->ri_dap_req != NULL)
  {
   f_kernel_mapping_system ();
   if (cuser_dap_ptr->ui_li_type.li_dap.dap_error == ErrCreateDB)
    break;
   curr_req_ptr->ri_dap_req = curr_req_ptr->ri_dap_req->dri_next_req;
  } /* end while */

 /* reset the number of requests */
 tran_info_ptr->ti_no_req = 0;

#ifdef EnExFlag
  printf ("Exit f_dbd_to_KMS\n");
#endif

} /* end f_dbd_to_KMS */

dapreqs_to_KMS()
        /* This routine causes the queries to be listed on the screen. */
        /* The selection menu is then displayed allowing any of the    */
        /* queries to be executed.                                     */

{
        int proceed;      /* boolean flag */
        int quit;         /* boolean flag */
        int num;

#ifdef EnExFlag
  printf ("Enter dapreqs_to_KMS\n");
#endif

 num = 0;
 list_dapreqs();
 proceed = TRUE;
 while (proceed == TRUE)
  {
   printf ("\nPick the number or letter of the action desired\n");
   printf ("\t(num) - execute one of the preceding DAPLEX requests\n");
   printf ("\t(d)   - redisplay the file of DAPLEX requests\n");
   printf ("\t(x)   - return to the previous menu\n");
   dap_info_ptr->dap_answer = get_ans(&num);
```

```
    switch (dap_info_ptr->dap_answer)
        {
         case 'n' : /* execute one of the requests */
           if (num > 0 && num <= tran_info_ptr->ti_no_req)
                {
                 find_dapreq (num);
            /* This is the default value for di_operation */
            /* If not a retrieve request, this value is reset */
            /* in kernel_mapping_system */
            dap_info_ptr->dap_operation = ExecRetReq;

                 quit = FALSE;
                 while (quit == FALSE)
                  {
                   f_kernel_mapping_system();
                   if (dap_info_ptr->dap_error == NOErr)
                       curr_req_ptr->ri_dap_req =
                          curr_req_ptr->ri_dap_req->dri_sub_req;
                   else
                       quit = TRUE;
                   if (curr_req_ptr->ri_dap_req == NULL)
                       quit = TRUE;
                  }

                 if (dap_info_ptr->dap_error == NOErr)
                  f_Kernel_Controller();
                } /* end if */
           else
                {
                 printf ("\nError - the DAPLEX request for the number you ");
            printf ("selected does not exist\n");
                 printf ("Please pick again\n");
                } /* end else */

         break;
         case 'd' : /* redisplay requests */
           list_dapreqs();
           break;
         case 'x' : /* exit to mode menu */
           proceed = FALSE;
         tran_info_ptr->ti_no_req = 0;
           break;
         default : /* user did not select a valid choice from the menu */
           printf ("\nError - invalid option selected\n");
           printf ("Please pick again\n");
           break;
      } /* end switch */

   } /* end while */

#ifdef EnExFlag
```

```
  printf ("Exit dapreqs_to_KMS\n");
#endif

} /* end dapreqs_to_KMS */

list_dapreqs()
        /* This routine actually prints the query list to the screen */

{
        struct  dap_req_info  *sub_req_ptr; /* ptr to a subrequest     */
        struct  temp_str_info *req_ptr;     /* ptr to a line of a query */
        int       i;                        /* the number of the query  */
        int     first_line;                 /* boolean flag             */
      FILE    *qry_fid;                     /* file id for query print file */


#ifdef EnExFlag
  printf ("Enter list_dapreqs\n");
#endif

 i = 1;
 qry_fid = fopen(QRYFName, "w");
 curr_req_ptr->ri_dap_req = first_req_ptr->ri_dap_req;

 /* loop and print the queries until there are no more */
 while (curr_req_ptr->ri_dap_req != NULL)
  {
   first_line = TRUE;
   fprintf (qry_fid,"\n");
   sub_req_ptr = curr_req_ptr->ri_dap_req;

   /* loop and print the subqueries until there are no more */
   while (sub_req_ptr != NULL)
       {
        req_ptr = sub_req_ptr->dri_in_req;
      while (req_ptr != NULL)
          {
         if (first_line == TRUE)
             {
              /* first line of a query so print the number of it first */
              fprintf (qry_fid, "\t%d\t %s\n", i, req_ptr->tsi_str);
              first_line = FALSE;
             } /* end if */
         else
             fprintf (qry_fid, "\t\t %s\n", req_ptr->tsi_str);
           req_ptr = req_ptr->tsi_next;
          } /* end while */

         sub_req_ptr = sub_req_ptr->dri_sub_req;
        } /* end while */
```

```
    curr_req_ptr->ri_dap_req = curr_req_ptr->ri_dap_req->dri_next_req;
    i++;
   } /* end while */

 fclose( qry_fid ); /* close the query file */

 moredapreqs( QRYFName ); /* print out the queries */

#ifdef EnExFlag
  printf ("Exit list_dapreqs\n");
#endif

} /* end list_dapreqs */

find_dapreq(num)
        int num;        /* specified query to be executed */
{
        int i;          /* counter */

        /* This function walks down the DAPLEX request list to the (num)th */
        /* query and leaves the current ptr pointing to that request */

#ifdef EnExFlag
  printf ("Enter find_dapreq\n");
#endif

  /* set the current ptr to the first ptr */
  curr_req_ptr->ri_dap_req = first_req_ptr->ri_dap_req;
  for (i = 1; i < num; i++)
    curr_req_ptr->ri_dap_req = curr_req_ptr->ri_dap_req->dri_next_req;

#ifdef EnExFlag
  printf ("Exit find_dapreq\n");
#endif

} /* end find_dapreq */

f_free_requests()
{

        struct  dap_req_info  *sub_req_ptr;

        /* This function frees all memory reserved by the transaction list */

#ifdef EnExFlag
  printf ("Enter f_free_requests\n");
#endif

  /* set the current ptr to the first ptr */
  curr_req_ptr->ri_dap_req = first_req_ptr->ri_dap_req;
```

```
  while (curr_req_ptr->ri_dap_req != NULL)
    {
     while (curr_req_ptr->ri_dap_req->dri_sub_req != NULL)
         {
          sub_req_ptr = curr_req_ptr->ri_dap_req->dri_sub_req;
          curr_req_ptr->ri_dap_req->dri_sub_req = sub_req_ptr->dri_sub_req;
          free (sub_req_ptr);
         } /* end while */

     curr_req_ptr->ri_dap_req = curr_req_ptr->ri_dap_req->dri_next_req;
     free (first_req_ptr->ri_dap_req);
     first_req_ptr->ri_dap_req = curr_req_ptr->ri_dap_req;
    } /* end while */

#ifdef EnExFlag
   printf ("Exit f_free_requests\n");
#endif

} /* end f_free_requests */

static moredapreqs(fname)
char  *fname;
{
  int   c,counter;
  char  ch[3];
  int   i,j;
  int   no_lines;
  FILE  *fid;


  fid = fopen(fname,"r");

  no_lines = 21;

  c = fgetc(fid);
  while(c != EOF)
    {
     counter = 0;
     while ((counter <= no_lines) && (c != EOF))
       {
        printf("%c",c);
        if (c == '\n')
          ++counter;
        c = fgetc(fid);
       }

     if (counter >= no_lines)
       {
        printf("\n[7;7m-- more --[0;0m");
        for(i = 0; ( (i <= 2) && ((ch[i] = getchar())!='\n') ); i++)
          ;
```

```
        if ((i==1) && (ch[0] == 'q'))
          return;
        else if (i == 1)
          {
            ch[1] = '\0';
            no_lines = str_to_num(ch) - 1;
          }
        else
          if (i==2)
            {
              ch[2] = '\0';
              no_lines = str_to_num(ch) - 1;
            }
          else
            no_lines = 21;

        printf("\n");
      } /* end if */
    } /* end while eof */

  fclose( fid );
} /* end more */
```

3. FILE : newuser.c

```
/* This proc allocates and initializes a node and structure of the user list */


#include <stdio.h>
#include "licommdata.def"
#include "struct.def"
#include "flags.def"
#include "dap.ext"

struct user_info *new_dap_user()
{
  struct user_info    *new_user_alloc(), /* ptr to new allocated user struct */
                      *user_ptr,         /* ptr to user struct */
                      *new_user_ptr;     /* ptr to new allocated user struct */
  struct dap_info     *dap_info_ptr;     /* temp ptr to dap_info struct      */
  struct curr_db_info *curr_db_info_ptr; /* temp ptr to curr_db_info struct  */
  struct file_info    *file_info_ptr;    /* temp ptr to file_info struct     */
  struct tran_info    *tran_info_ptr;    /* temp ptr to tran_info struct     */
  struct kc_ent_info  *kc_ptr;           /* temp ptr to kc_dap_info struct   */
  struct kfs_ent_info *kfs_ptr;          /* temp ptr to kfs_dap_info struct  */
  char  *var_str_alloc();

#ifdef EnExFlag
   printf ("Enter new_dap_user\n");
#endif
```

```
/* allocate a new user structure */
new_user_ptr = new_user_alloc();
/* set user_ptr to the head of the user list */
user_ptr = user_dap_head_ptr;
if (user_ptr == NULL)
 {
  /* user list is empty */
  user_dap_head_ptr = new_user_ptr;
  user_ptr = new_user_ptr;
 }
else
 {
  /* walk down user list to the end and append new user structure */
  while (user_ptr->ui_next_user != NULL)
   user_ptr = user_ptr->ui_next_user;
  user_ptr->ui_next_user = new_user_ptr;
  user_ptr = new_user_ptr;
 }

/* initialize complete new user structure */
get_new_user (user_ptr->ui_uid);
dap_info_ptr = &(user_ptr->ui_li_type.li_dap);
dap_info_ptr->dap_operation = 0;
dap_info_ptr->dap_answer = 0;
dap_info_ptr->dap_error = NOErr;
dap_info_ptr->dpi_ddl_files = NULL;
dap_info_ptr->dap_buff_count = 0;
curr_db_info_ptr = &(dap_info_ptr->dpi_curr_db);
strcpy (curr_db_info_ptr->cdi_dbname, DUMMYDBname);
curr_db_info_ptr->cdi_db.dn_fun = NULL;
curr_db_info_ptr->cdi_attr.an_nattr_ptr = NULL; /* ????? */
file_info_ptr = &(dap_info_ptr->dpi_file);
strcpy (file_info_ptr->fi_fname, DUMMYFname);
file_info_ptr->fi_fid = NULL;
tran_info_ptr = &(dap_info_ptr->dpi_dml_tran);
dap_info_ptr->dpi_kms_data.ki_n_kms = NULL;
kfs_ptr = &(dap_info_ptr->dpi_kfs_data.kfsi_ent);

/* kri_response is now allocated here rather than in procedure    */
/* dap_chk_responses_left in kc.c                                 */
kfs_ptr->khi_response = var_str_alloc(1024);
kfs_ptr->khi_curr_pos = 0;
kfs_ptr->khi_res_len = 0;
kc_ptr = &(dap_info_ptr->dpi_kc_data.kci_e_kc);
kc_ptr->temp = 0;


#ifdef EnExFlag
  printf ("Exit new_dap_user\n");
#endif
```

```
 return (new_user_ptr);

} /* end new_dap_user */
```

4. FILE : readrtnes.c

```
#include <stdio.h>
#include <ctype.h>
#include "flags.def"
#include "licommdata.def"
#include "struct.def"
#include <strings.h>
#include "dap.ext"
#include "lil.ext"

f_read_transaction_file()
{
        int   open_flag;  /* boolean flag */
        int   i;     /* counter      */

        /* This routine opens a DAPLEX dbd/request file and reads the */
        /* dbds/requests into the request list.                       */

#ifdef EnExFlag
  printf ("Enter f_read_transaction_file\n");
#endif

 open_flag = FALSE;
 printf ("[7;7m\nWhat is the name of the DBD/REQUEST file ---->[0;0m ");

 /* open the file */
 while (open_flag == FALSE)
  {
   readstr (stdin, dap_info_ptr->dpi_file.fi_fname);
   printf ("\n\n");
   if ((dap_info_ptr->dpi_file.fi_fid =
                fopen (dap_info_ptr->dpi_file.fi_fname, "r")) == NULL)
       {
         printf ("\nUnable to open file %s\n",dap_info_ptr->dpi_file.fi_fname);
         ring_the_bell();
         printf ("Please reenter valid filename\n");
      printf ("[7;7m");
      printf ("\nWhat is the name of the DBD/REQUEST file --->[0;0m ");
       } /* end if */
   else
       open_flag = TRUE;

  } /* end while */

 /* now read in the transactions */
```

```
 f_read_file();

#ifdef EnExFlag
  printf ("Exit f_read_transaction_file\n");
#endif

} /* end f_read_transaction_file */

f_read_file()
{
    struct temp_str_info *new_t_ptr,  /* ptrs to linked list of 80 col input */
                         *curr_t_ptr,
                          *head_t_ptr,
                                *f_rd_temp_str_info();
    struct dap_req_info  *new_req_ptr,   /* ptrs to request list */
                          *curr_req_ptr,
                          *sub_req_ptr,
                          *dap_req_info_alloc();
    int    i,
             first_subreq,  /* boolean flag */
           first_req,     /* boolean flag */
             first_line,    /* boolean flag */
             length_so_far, /* length of a single transaction */
             EOF_flag,      /* boolean flag */
           EOR_flag,      /* boolean flag */
             EOS_flag;      /* boolean flag */
    char   *var_str_alloc();

          /* This routine reads a file of transactions into */
          /* the user's request list structure.                  */

#ifdef EnExFlag
  printf ("Enter f_read_file\n");
#endif

 first_req = TRUE;
 EOF_flag = FALSE;

 /* create the request list from the inner loop's line list */
 while (EOF_flag == FALSE)
  {
   EOS_flag = FALSE;
   EOR_flag = FALSE;
   length_so_far = 0;
   first_line = TRUE;
   first_subreq = TRUE;

   /* create a line list for each request read.  */
   /* each node represents a line of the request */
   while (EOR_flag == FALSE)
    {
```

```
 /* allocate a line */
 new_t_ptr = f_rd_temp_str_info (&EOS_flag, &EOR_flag, &EOF_flag);
 if (new_t_ptr != NULL)
  {
   length_so_far = length_so_far + strlen (new_t_ptr->tsi_str);
   if (first_line == TRUE)
       {
        /* line is the first on the list so set appropriate ptrs */
        head_t_ptr = new_t_ptr;
        curr_t_ptr = new_t_ptr;
        first_line = FALSE;
       } /* end if */
   else
       {
        /* link line to the rest of the line list */
        curr_t_ptr->tsi_next = new_t_ptr;
        curr_t_ptr = new_t_ptr;
       } /* end else */

} /* end if */

 else

/* check for no input situation */
if (EOS_flag == FALSE && first_line == TRUE && EOF_flag == TRUE)
    {
     dap_info_ptr->dap_error = ErrReadFile;
     printf ("WARNING - number of requests read = 0!\n\n");
    }
else
    {
     /* allocate a request structure */
   new_req_ptr = dap_req_info_alloc();
   /* store head_t_ptr as the input request */
   new_req_ptr->dri_in_req = head_t_ptr;
   new_req_ptr->dri_req = var_str_alloc(length_so_far + 1);
   new_req_ptr->dri_req[0] = '\0';
   curr_t_ptr = head_t_ptr;

     /* concatenate line list to form a request node */
   while (curr_t_ptr != NULL)
       {
        strcat (new_req_ptr->dri_req, curr_t_ptr->tsi_str);
        curr_t_ptr = curr_t_ptr->tsi_next;
       } /* end while */

     /* capitalize the request */
   to_caps (new_req_ptr->dri_req);
   new_req_ptr->dri_req_len = length_so_far;
        new_req_ptr->dri_sub_req = NULL;
   new_req_ptr->dri_next_req = NULL;
```

```
            if (EOS_flag == TRUE)
              {
               if (first_subreq == TRUE)
                 {
                  sub_req_ptr = new_req_ptr;
                  curr_req_ptr = new_req_ptr;
                  first_subreq = FALSE;
                 } /* end if */
               else
                 {
                  curr_req_ptr->dri_sub_req = new_req_ptr:
                  curr_req_ptr = new_req_ptr;
                 } /* end else */

               length_so_far = 0;
               first_line = TRUE;
              } /* end if */
          } /* end else */


  } /* end while EOR_flag = FALSE */


if (first_req == TRUE)
  {
      if (EOS_flag == TRUE)
        {
       /* request is the first on the sublist so set appropriate ptrs */
          tran_info_ptr->ti_first_req.ri_dap_req = sub_req_ptr;
          tran_info_ptr->ti_curr_req.ri_dap_req  = sub_req_ptr;
        } /* end if */
      else
        {
       /* request is the first on the list so set appropriate ptrs */
          tran_info_ptr->ti_first_req.ri_dap_req = new_req_ptr;
          tran_info_ptr->ti_curr_req.ri_dap_req  = new_req_ptr;
        } /* end else */
    ++tran_info_ptr->ti_no_req;
      first_req = FALSE;
  } /* end if first_req = TRUE */
else
     {
    if (EOS_flag == TRUE)
        {
         tran_info_ptr->ti_curr_req.ri_dap_req->dri_next_req = sub_req_ptr;
         tran_info_ptr->ti_curr_req.ri_dap_req = sub_req_ptr;
        } /* end if */
      else
        {
         tran_info_ptr->ti_curr_req.ri_dap_req->dri_next_req = new_req_ptr;
         tran_info_ptr->ti_curr_req.ri_dap_req = new_req_ptr;
```

```
            } /* end else */
        ++tran_info_ptr->ti_no_req;
         } /* end else first_req = FALSE */

    } /* end while EOF_flag = FALSE */

#ifdef EnExFlag
  printf ("Exit f_read_file\n");
#endif

} /* end f_read_file */

f_read_terminal()
{

        /* This function prompts the user to input DAPLEX  requests */
        /* from their terminal                                      */

#ifdef EnExFlag
  printf ("Enter f_read_terminal\n");
#endif

  /* set input device to be the terminal */
  dap_info_ptr->dpi_file.fi_fid = stdin:
  printf ("\nPlease enter your transactions one at a time.\n");
  printf ("You may have multiple lines per transaction.\n");
  printf ("Each transaction must be separated by a line that\n");
  printf ("\tonly contains the character '@'.\n");
  printf ("If you have multiple requests per transactions, seperate\n");
  printf ("\tthem by the character '!'.\n");
  printf ("After the last transaction, the last line must consist only\n");
  printf ("\tof the '$' character to signal end-of-file.\n\n\n");
  printf ("[7;7m Input the transactions on the following lines :[0;0m\n\n");

  /* now read in the transactions */
  f_read_file();

#ifdef EnExFlag
  printf ("Exit f_read_terminal\n");
#endif

} /* end f_read_terminal */

static  struct temp_str_info *f_rd_temp_str_info(EOS_flag, EOR_flag, EOF_flag)
        int    *EOS_flag,     /* boolean flags */
           *EOR_flag,
             *EOF_flag;

{
        struct temp_str_info *temp_str_info_alloc(),
                             *temp_ptr;  /* ptrs to new temp_str_info structs */
```

```
    int         i;                          /* counter */

      /* This routine fills an allocated line list node */
      /* and sends back a pointer to the node.          */

#ifdef EnExFlag
  printf ("Enter f_rd_temp_str_info\n");
#endif

  /* set a ptr to the allocated line */
  temp_ptr = temp_str_info_alloc();

  /* now read in a line of input */
  readstr (dap_info_ptr->dpi_file.fi_fid, temp_ptr->tsi_str);
  for (i = 0; temp_ptr->tsi_str[i] != '\0'; i++)
    ;
  temp_ptr->tsi_str[i++] = ' ';
  temp_ptr->tsi_str[i] = '\0';
  temp_ptr->tsi_next = NULL;

  /* check for end-of-subrequest ('!') */
  if (temp_ptr->tsi_str[0] == EOSubrequest)
   {
    *EOS_flag = TRUE;

#ifdef EnExFlag
   printf ("Exit1 f_rd_temp_str_info\n");
#endif

    return (NULL);
   } /* end if */

  else
   /* check for end-of-request ('@') */
   if (temp_ptr->tsi_str[0] == EORequest)
    {
     *EOR_flag = TRUE;

#ifdef EnExFlag
  printf ("Exit2 f_rd_temp_str_info\n");
#endif

      return (NULL);
    } /* end if */
   else
    /* check for end-of-file ('$') */
    if (temp_ptr->tsi_str[0] == EOFile)
     {
         *EOR_flag = TRUE;
         *EOF_flag = TRUE;
```

```
#ifdef EnExFlag
  printf ("Exit3 f_rd_temp_str_info\n");
#endif

        return (NULL);
       } /* end else if */
     else

#ifdef EnExFlag
  printf ("Exit4 f_rd_temp_str_info\n");
#endif

      return (temp_ptr);

} /* end f_rd_temp_str_info */
```

5. FILE : buildddl.c

```
#include <stdio.h>
#include "licommdata.def"
#include "struct.def"
#include "flags.def"
#include "lil.ext"



f_build_ddl_files()
  /* This routine is used to create the MBDS template and descriptor files */
{
 struct  ddl_info  *ddl_info_alloc();

#ifdef EnExFlag
        printf("Enter f_build_ddl_file\n");
#endif

 if (dap_info_ptr -> dpi_ddl_files == NULL)
    dap_info_ptr -> dpi_ddl_files = ddl_info_alloc();

 f_build_template_file();
 f_build_desc_file();

#ifdef EnExFlag
        printf("Exit f_build_ddl_file\n");
#endif

/*  f_build_desc_file();*/
}
```

```
f_build_template_file()
{
  /* This routine builds the MBDS template file for a new daplex  */
  /* database that was just created.  */

  struct fun_dbid_node   *db_ptr;
  struct ent_node        *ent_ptr;
  struct gen_sub_node    *gen_ptr;
  struct function_node   *funct_ptr;
  struct file_info       *f_ptr;
  char                   temp_str[NUMDIGIT + 1];
  char                   get_fun_type();

#ifdef EnExFlag
        printf("Enter f_build_template_file\n");
#endif

  /* Begin by setting the pointers to the dap_info data structure */
  /* that is maintained for each user of the system.              */
  db_ptr = dap_info_ptr->dpi_curr_db.cdi_db.dn_fun;
  f_ptr  = &(dap_info_ptr->dpi_ddl_files->ddli_temp);

  /* Next, copy the filename where the MBDS template information will */
  /* be stored.  This filename is constant and was obtained from      */
  /* licommdata.def.  */
  strcpy(f_ptr->fi_fname, FTEMPFname);

  /*  Next, open the template File to be created:                     */
  if ((f_ptr->fi_fid = fopen(f_ptr->fi_fname,"w")) == NULL)
        {
        printf("Unable to open %s\n", FTEMPFname);
        ring_the_bell();

#ifdef EnExFlag
        printf("Exit1 f_build_template_file\n");
#endif
        return;
        };

  /*  Next, write out the database name & number of files :           */
  fprintf(f_ptr->fi_fid, "%s\n", db_ptr->fdn_name);
  num_to_str(db_ptr->fdn_num_ent + db_ptr->fdn_num_gen, temp_str);
  fprintf(f_ptr->fi_fid, "%s\n", temp_str);

  /*  Next, set the pointer to the first entity:                      */
  ent_ptr = db_ptr->fdn_entptr;

  /*  While there are more entities to process, write out the number  */
  /*  of functions (+2 for the attribute "FILE" and the key value attribute) */
  /*  and the entity name:      */
  while (ent_ptr != NULL)
```

```
    {
    num_to_str((ent_ptr->en_num_funct + 2), temp_str);
    fprintf(f_ptr->fi_fid, "%s\n", temp_str);
    fprintf(f_ptr->fi_fid, "%s\n", ent_ptr->en_name);

    /* Print out the  onstant attribute "FILE s" and key value attribute  */
    fprintf(f_ptr->fi_fid, "FILE s\n");
    fprintf(f_ptr->fi_fid, "%s i\n", to_caps(ent_ptr->en_name));

    /* Now, set the pointer to the first function:                        */
    funct_ptr = ent_ptr->en_ftnptr:
    wr_all_funct_attr(f_ptr->fi_fid, funct_ptr);

    /* set the pointer to the next entity:                                */
    ent_ptr = ent_ptr->en_next_ent;

    }  /* end while ent_ptr */

  /* Next, set the pointer to the first gen sub node :                    */
  gen_ptr = db_ptr->fdn_subptr:

  /* While there are more sub nodes to process, write out the number      *
  /* of functions (+2 for the attribute "FILE" and the key value attribute) *
  /* and the entity name:      */
  while (gen_ptr != NULL)
    {
    num_to_str((gen_ptr->gsn_num_funct + 2), temp_str);
    fprintf(f_ptr->fi_fid, "%s\n", temp_str);
    fprintf(f_ptr->fi_fid, "%s\n", gen_ptr->gsn_name);

    /* Print out the constant attribute "FILE s" and key value attribute  */
    fprintf(f_ptr->fi_fid, "FILE s\n");
    fprintf(f_ptr->fi_fid, "%s i\n", to_caps(gen_ptr->gsn_name));

    /* Now, set the pointer to the first function:                        */
    funct_ptr = gen_ptr->gsn_ftnptr;
    wr_all_funct_attr(f_ptr->fi_fid, funct_ptr);

    /* set the pointer to the next gen sub node:                          */
    gen_ptr = gen_ptr->gsn_next_genptr;

    }  /* end while gen_ptr */

  /* Finally, close out the file and exit this routine:                   */
  fclose(f_ptr->fi_fid);

#ifdef EnExFlag
        printf("Exit2 f_build_template_file\n");
#endif
}
```

```
wr_all_funct_attr(fid, funct_ptr)
  FILE   *fid;
  struct  function_node   *funct_ptr;
  {

#ifdef EnExFlag
        printf("Enter wr_all_funct_attr\n");
#endif

    /*  While there are more attributes to process,                    */
    /*  print out attr. name & type:        */
    while (funct_ptr != NULL)
      {
      fprintf(fid, "%s %c\n", funct_ptr->fn_name, get_fun_type(funct_ptr));

      /*  Set the pointer to the next function:                        */
      funct_ptr = funct_ptr->next;

      }  /* end while funct_ptr */

#ifdef EnExFlag
        printf("Exit wr_all_funct_attr\n");
#endif
  }

char get_fun_type(fptr)
        struct   function_node   *fptr;
        { /* begin get_fun_type */
                char    fun_type;

#ifdef EnExFlag
        printf("Enter get_fun_type\n");
#endif

         switch (fptr->fn_type)
           {
                case 'i' : ;
                case 'f' : ;
                case 's' : fun_type = fptr->fn_type;
                           break;
                case 'b' : fun_type = 'i';
                           break;
                case 'e' : if (fptr -> fn_entptr != NULL ||
                               fptr -> fn_subptr != NULL)
                               fun_type = 'i';
                           else if (fptr -> fn_nonentptr != NULL)
                               fun_type = fptr -> fn_nonentptr -> enn_type;
                           else if (fptr -> fn_nonsubptr != NULL)
                               fun_type = fptr -> fn_nonsubptr -> snn_type;
                           else if (fptr -> fn_nonderptr != NULL)
                               fun_type = fptr -> fn_nonderptr -> dnn_type;
```

```
                        if (fun_type =='e')     /* still not i or f type */
                           fun_type = 's';
                        break;
          }  /* end switch */

#ifdef EnExFlag
        printf("Exit get_fun_type\n");
#endif
        return(fun_type);

        } /* end get_fun_type */
```

6. FILE : builddesc.c

```
#include <stdio.h>
#include "licommdata.def"
#include "struct.def"
#include "flags.def"
#include "lil.ext"
#include "dap.ext"
#include <strings.h>
#include <ctype.h>

f_build_desc_file()

  {
  /* This routine builds the Descriptor File to be used by the MBDS in the   */
  /* creation of indexing clusters:                                          */

  struct   fun_dbid_node    *db_ptr;          /*  database pointer      */
  struct   ent_node         *ent_ptr;         /*  entity node ptr       */
  struct   gen_sub_node     *gen_ptr;         /*  gen sub node ptr      */
  struct   descriptor_node  *desc_head_ptr,    /*  pointers to Desc_node..*/
                            *ask_all_fun_nodes();
  struct   file_info        *f_ptr;           /*  File pointer          */
  int                       index,
                            str_len;

#ifdef EnExFlag
        printf("Enter f_build_desc_file\n");
#endif

  /*  Begin by setting the pointers to the dap_info data structure that is   */
  /*  maintained for each user of the system:                                */
  db_ptr = dap_info_ptr->dpi_curr_db.cdi_db.dn_fun;
  f_ptr = &(dap_info_ptr->dpi_ddl_files->ddli_desc);

  /*  Next, copy the filename where the MBDS Descriptor File information     */
  /*  will be stored.  This filename is Constant, and was obtained from      */
  /*  licommdata.def:                                                        */
```

```
strcpy(f_ptr->fi_fname, FDESCFname);

/*  Now, open the Descriptor File to be created:                    */
f_ptr->fi_fid = fopen(f_ptr->fi_fname, "w");

/*  The next step is to traverse the Linked List of entities in the data-  */
/*  base.  There are two reasons for doing so:  First, to write the        */
/*  entity Names to the Descriptor File as EQUALITY Descriptors; this is   */
/*  done automatically with any Daplex Database, is a necessary element    */
/*  of any Descriptor File created from such a Database, and requires      */
/*  no user involvement.  Second, it allows us to present the Entity       */
/*  Names (without their respective Attributes) to the User, as a memory   */
/*  jog:                                                                   */

system("clear");
fprintf(f_ptr->fi_fid, "%s\n", db_ptr->fdn_name);
fprintf(f_ptr->fi_fid, "template C s\n");
printf("\nThe following are the Entities in the ");
printf("%s Database:\n\n", db_ptr->fdn_name);

ent_ptr = db_ptr->fdn_entptr;

/*  Traverse all entity nodes                                          */
while (ent_ptr != NULL)
  {
  fprintf(f_ptr->fi_fid, "! %c", ent_ptr->en_name[0] );
  str_len = strlen( ent_ptr->en_name );
  for(index = 1; index < str_len: index++)
    if (isupper(ent_ptr->en_name[index]))
        fprintf(f_ptr->fi_fid, "%c", tolower( ent_ptr->en_name[index] ));
    else
        fprintf(f_ptr->fi_fid, "%c", ent_ptr->en_name[index]);
  fprintf(f_ptr->fi_fid, "\n");
  printf("\n\t%s", ent_ptr->en_name);
  ent_ptr = ent_ptr->en_next_ent;
  }  /*  End "while (ent_ptr != NULL)"  */

gen_ptr = db_ptr->fdn_subptr;

/*  Traverse all gen sub nodes                                         */
while (gen_ptr != NULL)
  {
  fprintf(f_ptr->fi_fid, "! %c", gen_ptr->gsn_name[0] );
  str_len = strlen( gen_ptr->gsn_name );
  for(index = 1; index < str_len; index++)
    if (isupper(gen_ptr->gsn_name[index]))
        fprintf(f_ptr->fi_fid, "%c", tolower( gen_ptr->gsn_name[index] ));
    else
        fprintf(f_ptr->fi_fid, "%c", gen_ptr->gsn_name[index]);
  fprintf(f_ptr->fi_fid, "\n");
  printf("\n\t%s", gen_ptr->gsn_name);
```

```
    gen_ptr = gen_ptr->gsn_next_genptr;
    }  /*  End "while (gen_ptr != NULL)"  */

  /*  Each Descriptor Block must be followed by the "@" sign:          */
  fprintf(f_ptr->fi_fid, "@\n");

  /*  Now, inform the user of the procedure that must be followed to create  */
  /*  the Descriptor File:                                             */
  printf("\n\nBeginning with the first Entity, we will present each");
  printf("\nfunction of the entity.  You will be prompted as to whether");
  printf("\nyou wish to include that function as an Indexing Attribute.");
  printf("\nand, if so, whether it is to be indexed based on strict");
  printf("\nEQUALITY, or based on a RANGE OF VALUES.");
  printf("\n\nStrike RETURN when ready to continue.");
  dap_info_ptr->dap_answer = get_ans(&index);

  /*  Initialize the pointer to a Linked List that will hold the results   */
  /*  of the Descriptor Values, then return to the first entity of the     */
  /*  database and begin cycling through the individual attributes:        */
  desc_head_ptr = NULL;

  ent_ptr = db_ptr->fdn_entptr;
  while (ent_ptr != NULL)
    {
    desc_head_ptr =
     ask_all_fun_nodes(desc_head_ptr, ent_ptr->en_name, ent_ptr->en_ftnptr);

    ent_ptr = ent_ptr->en_next_ent;
    }  /*  End while */

  gen_ptr = db_ptr->fdn_subptr;
  while (gen_ptr != NULL)
    {
    desc_head_ptr =
     ask_all_fun_nodes(desc_head_ptr, gen_ptr->gsn_name, gen_ptr->gsn_ftnptr);
    gen_ptr = gen_ptr->gsn_next_genptr;
    }  /*  End while */

  /*  Now, we will traverse the Linked List of Descriptor Attributes and   */
  /*  Values which was created, writing them to our Descriptor File:       */

    wr_all_desc_values(f_ptr->fi_fid, desc_head_ptr);
    fclose(f_ptr->fi_fid);

#ifdef EnExFlag
        printf("Exit f_build_desc_file\n");
#endif
  }


struct descriptor_node *ask_all_fun_nodes(desc_head_ptr, en_name, funct_ptr)
```

```
  struct  descriptor_node   *desc_head_ptr;
  char    en_name[ENLength + 1];
  struct  function_node     *funct_ptr;
  {
  struct  descriptor_node   *desc_node_ptr,
                    *descriptor_node_alloc();  /* Allocates Nodes  */
  struct   value_node       *valuenode_ptr;     /* points to Value Node   */
  int                  num,              /*                        */
                       found,            /*  Boolean flag          */
                       goodanswer;       /*  Boolean flag          */

#ifdef EnExFlag
        printf("Enter ask_all_fun_nodes\n");
#endif

  while (funct_ptr != NULL)
   {
   if (funct_ptr->fn_entptr == NULL && funct_ptr->fn_subptr == NULL)
     {

     system("clear");
     printf("Entity name:  %s\n", en_name):
     printf("Function Name: %s\n\n", funct_ptr->fn_name);

     /*  Now. traverse the Attribute linked list that is being created,   */
     /*  to see if the current Attribute has already been established as  */
     /*  a Descriptor Attribute.  If so. offer the user the opportunity   */
     /*  to add additional EQUALITY or RANGE OF VALUE values; otherwise,  */
     /*  offer the user the opportunity to establish this as a Descriptor */
     /*  Attribute:                                                       */
     desc_node_ptr = desc_head_ptr;
     found = FALSE;
     while ((desc_node_ptr != NULL) && (found == FALSE))
       {
       if (strcmp(funct_ptr->fn_name, desc_node_ptr->attr_name) == 0)
         {

         /*  The Attribute HAS already been chosen as a Descriptor.     */
         /*  Allow the user the option of adding additional Descriptor  */
         /*  values, after listing those already entered:               */

         printf("\nThis Attribute has been chosen as an Indexing Attribute.\n");
         printf("The following are the values that have been specified:\n\n");
         found = TRUE;
         valuenode_ptr = desc_node_ptr->first_value_node;
         while (valuenode_ptr != NULL)
           {
           if (desc_node_ptr->descriptor_type == 'A')
             printf("\t%s %s\n", valuenode_ptr->value1,
                              valuenode_ptr->value2);
           else
```

```
          printf("\t%s\n", valuenode_ptr->value2);
        valuenode_ptr = valuenode_ptr->next_value_node;
        }  /* End "while (valuenode_ptr != NULL)" */
      printf("\nDo you wish to add more ");
      if (desc_node_ptr->descriptor_type == 'A')
        printf("RANGE");
      else
        printf("EQUALITY");
      printf(" values?  (y or n)\n");
      dap_info_ptr->dap_answer = get_ans(&num);
      if ((dap_info_ptr->dap_answer == 'y') ||
         (dap_info_ptr->dap_answer == 'Y'))

        /* The user DOES wish to add more descriptors to the      */
        /* currently existing list:                               */
        {
        if (desc_node_ptr->descriptor_type == 'A')
          build_RAN_descrip(desc_node_ptr, funct_ptr->fn_total_length);
        else
          build_EQ_descrip(desc_node_ptr, funct_ptr->fn_total_length);
        }  /* End "if ((dap_info_ptr->dap_answer == 'y') ||
                   (dap_info_ptr->dap_answer == 'Y'))        */
    }  /* End "if (strcmp(...) == 0)" */
  desc_node_ptr = desc_node_ptr->next_desc_node;
  }  /* End "while ((desc_node_ptr != NULL) && (found..))" */

if (found == FALSE)

  /* The Attribute has NOT previously been chosen as a Descriptor. */
  /* Allow the user the option of making this a Descriptor Attri-  */
  /* bute, with appropriate Descriptor Values:                     */
  {
  printf("\nDo you wish to install this function as an ");
  printf("Indexing Attribute?\n\n");
  printf("\t(n) - no; continue with next Attribute/Relation\n");
  printf("\t(e) - yes; establish this as an EQUALITY Attribute\n");
  printf("\t(r) - yes; establish this as a RANGE Attribute\n");
  goodanswer = FALSE;
  while (goodanswer == FALSE)
    {
    dap_info_ptr->dap_answer = get_ans(&num);

    switch(dap_info_ptr->dap_answer)
      {
      case 'n':  /* User does NOT want to use this as an        */
                 /* Indexing (Descriptor) Attribute:            */
                 goodanswer = TRUE;
                 break;

      case 'e':  /* User wants to use this as an EQUALITY       */
                 /* Attribute:                                  */
```

```
                goodanswer = TRUE;
                desc_node_ptr = descriptor_node_alloc();
                desc_node_ptr->next_desc_node = desc_head_ptr;
                desc_head_ptr = desc_node_ptr;
                strcpy(desc_node_ptr->attr_name, funct_ptr->fn_name);
                desc_node_ptr->descriptor_type = 'B';
                desc_node_ptr->value_type = get_fun_type(funct_ptr);
                        desc_node_ptr->first_value_node = NULL;
                        build_EQ_descrip(desc_node_ptr,
                             funct_ptr->fn_total_length);
                break;

        case 'r':  /*  User wants to use this as a RANGE Attribute: */
                goodanswer = TRUE;
                desc_node_ptr = descriptor_node_alloc();
                desc_node_ptr->next_desc_node = desc_head_ptr;
                desc_head_ptr = desc_node_ptr;
                strcpy(desc_node_ptr->attr_name, funct_ptr->fn_name);
                desc_node_ptr->descriptor_type = 'A';
                desc_node_ptr->value_type = get_fun_type(funct_ptr);
                        desc_node_ptr->first_value_node = NULL;
                build_RAN_descrip(desc_node_ptr,
                             funct_ptr->fn_total_length);
                break;

        default:   /*  User did not select a valid choice:        */
                printf("\nError - Invalid operation selected;\n");
                printf("Please pick again\n");
                break;

        }  /*  End Switch  */

      }  /*  End "While (goodanswer = FALSE)"  */

    }  /*  End "if (found == FALSE)"  */

  }  /* End "if (funct_ptr->fn_entptr ...." */

  funct_ptr = funct_ptr->next;
 }  /*  End "while (funct_ptr != NULL)"  */

#ifdef EnExFlag
        printf("Exit ask_all_fun_nodes\n");
#endif

 return(desc_head_ptr);

}
```

7. FILE : buildcomm.c

```
#include <stdio.h>
#include "licommdata.def"
#include "flags.def"
#include <ctype.h>
#include <strings.h>

build_EQ_descrip(desc_node_ptr, attr_length)

  struct    descriptor_node      *desc_node_ptr;
  int                        attr_length;

  {
  /*  This routine builds the EQUALITY Descriptor list for the current      */
  /*  Attribute:                                                            */


  int                  end_routine:          /*  Boolean flag   */
  int                  index:              /*  Loop Index     */
  int                  loop_count:           /*  Loop Index     */
  int                  val_count:            /*  Loop Index     */
  int                  str_len;            /*  Length of input */
  char                  *temp_value:          /*  Holds answer   */
  char                  *var_str_alloc();     /*  Allocates Str. */
  struct   value_node       *valuenode_ptr,     /* Points to Value Node   */
                      *value_node_alloc(); /* Allocates Value Nodes  */

#ifdef EnExFlag
        printf("Enter build_EQ_descrip\n");
#endif

  /*  Repetitively offer the user the opportunity to create EQUALITY de-    */
  /*  scriptors for the current Attribute, halting when the user enters     */
  /*  an empty carriage return ("<CR>").                                    */

  temp_value = var_str_alloc(attr_length + 1);
  end_routine = FALSE;
  while (end_routine == FALSE)
    {
    printf("\nEnter EQUALITY match value, or <CR> to exit: ");
    readstr(stdin, temp_value);
    str_len = strlen( temp_value );
    for (index = 0; temp_value[index] == ' '; index++)
      ;
    if ( str_len != index )
      {
      valuenode_ptr = value_node_alloc(attr_length);
      valuenode_ptr->next_value_node = desc_node_ptr->first_value_node;
      desc_node_ptr->first_value_node = valuenode_ptr;
      strcpy(valuenode_ptr->value1, "!");
```

```
      /* Convert first character in temp_value to upper case if nec.  */
      if (islower(temp_value[index]))
        temp_value[index] = toupper( temp_value[index] );

      /* Convert remaining chars in temp_value to lower case if nec. */
      for( loop_count = index + 1; loop_count <= str_len; loop_count++)
       if (isupper(temp_value[loop_count]))
         temp_value[loop_count] = tolower( temp_value[loop_count] );

      /* Store temp_value into value2 from index to end of string. */
      val_count = 0;
      for( loop_count = index; loop_count <= str_len; loop_count++)
       valuenode_ptr->value2[val_count++] = temp_value[loop_count];
      valuenode_ptr->value2[val_count] = '\0';
      }
    else
      end_routine = TRUE;
    }
  free(temp_value);

#ifdef EnExFlag
        printf("Exit build_EQ_descrip\n");
#endif
  }

build_RAN_descrip(desc_node_ptr, attr_length)

  struct    descriptor_node     *desc_node_ptr;
  int                       attr_length;

  {
  /*  This routine builds the RANGE OF VALUEs Descriptor list for the      */
  /*  current Attribute:                                                   */


  int                  end_routine;          /*  Boolean flag      */
  int                  good_upper_value;     /*  Boolean flag      */
  int                  index;              /*  Loop Index        */
  int                  loop_count;           /*  Loop Index     */
  int                  val_count;            /*  Loop Index     */
  int                  str_len;            /*  Length of input    */
  struct   value_node      *valuenode_ptr,     /* Points to Value Node   */
                      *value_node_alloc(); /* Allocates Value Nodes  */
  char                  *temp_value;             /*  Holds answer   */
  char                  *var_str_alloc();        /*  Allocates Str. */

#ifdef EnExFlag
        printf("Enter build_RAN_descrip\n");
#endif
```

```
/*  Repetitively offer the user the opportunity to create RANGE OF VALUE  */
/*  Descriptors for the current Attribute, halting when the user enters   */
/*  an empty carriage return ("<CR>").                                    */

temp_value = var_str_alloc(attr_length + 1);
end_routine = FALSE;
while (end_routine == FALSE)
  {
  printf("\nEnter Lower Bound, or <CR> to exit: ");
  readstr(stdin. temp_value);
  str_len = strlen( temp_value );
  for (index = 0; temp_value[index] == ' '; index++)
    ;
  if ( str_len != index )
    {
    valuenode_ptr = value_node_alloc();
    valuenode_ptr->next_value_node = desc_node_ptr->first_value_node;
    desc_node_ptr->first_value_node = valuenode_ptr;

    /* Convert first character in temp_value to upper case if nec.  */
    if (islower(temp_value[index]))
      temp_value[index] = toupper( temp_value[index] );

    /* Convert remaining chars in temp_value to lower case if nec. */
    for( loop_count = index - 1; loop_count <= str_len; loop_count++)
     if (isupper(temp_value[loop_count]))
       temp_value[loop_count] = tolower( temp_value[loop_count] );

    /* Store temp_value into value1 from index to end of string. */
    val_count = 0;
    for( loop_count = index; loop_count <= str_len; loop_count++)
     valuenode_ptr->value1[val_count++] = temp_value[loop_count];
    valuenode_ptr->value1[val_count] = '\0';

    good_upper_value = FALSE;
    while (good_upper_value == FALSE)
      {
      printf("\nEnter Upper Bound:");
      readstr(stdin. temp_value);
      str_len = strlen(temp_value);
      for (index = 0; temp_value[index] == ' '; index++)
        ;
      if (str_len != index)
        {
        /* Convert first character in temp_value to upper case if nec. */
        if (islower(temp_value[index]))
          temp_value[index] = toupper( temp_value[index] );

        /* Convert remaining chars in temp_value to lower case if nec. */
        for( loop_count = index + 1; loop_count <= str_len; loop_count++)
         if (isupper(temp_value[loop_count]))
```

```
            temp_value[loop_count] = tolower( temp_value[loop_count] );

          /* Store temp_value into value1 from index to end of string. */
          val_count = 0;
          for( loop_count = index; loop_count <= str_len; loop_count++)
            valuenode_ptr->value2[val_count++] = temp_value[loop_count];
          valuenode_ptr->value2[val_count] = '\0';

          good_upper_value = TRUE;
          }
        else
          printf("\nYou must supply a non-blank Upper Bound.\n");
        }
      }
    else
      end_routine = TRUE;
    }
#ifdef EnExFlag
        printf("Exit build_RAN_descrip\n");
#endif
  }


wr_all_desc_values(fid, desc_head_ptr)

/* This routine traverse the linked list of descriptor attributes      */
/* and write all descriptor values out to the Descriptor file          */

  FILE   *fid;
  struct descriptor_node     *desc_head_ptr;
  {
  struct descriptor_node     *desc_node_ptr;
  struct value_node          *valuenode_ptr;    /* points to Value Node   */

#ifdef EnExFlag
        printf("Enter wr_all_desc_values\n");
#endif

  desc_node_ptr = desc_head_ptr;

  while (desc_node_ptr != NULL)
    {
    if (desc_node_ptr->first_value_node != NULL)
      {
      fprintf(fid, "%s %c %c\n", desc_node_ptr->attr_name,
                          desc_node_ptr->descriptor_type,
                          desc_node_ptr->value_type);
      valuenode_ptr = desc_node_ptr->first_value_node;
      while (valuenode_ptr != NULL)
        {
        fprintf(fid,"%s %s\n", valuenode_ptr->value1,
```

```
                        valuenode_ptr->value2);
          valuenode_ptr = valuenode_ptr->next_value_node;
          } /* End "while (valuenode_ptr != NULL)" */

        fprintf(fid,"@\n");
        } /* End "if (desc_node_ptr->first_value_node != NULL) */

      desc_node_ptr = desc_node_ptr->next_desc_node;

      } /* End "while (desc_node_ptr != NULL)" */

    fprintf(fid, "$\n");

#ifdef EnExFlag
          printf("Exit wr_all_desc_values\n");
#endif
    }
```

APPENDIX C

THE KMS MODULE

1. FILE : d.y

```
%{

#include <stdio.h>
#include <strings.h>
#include "licommdata.def"
#include "struct.def"
#include "dap.ext"
#include "lil.ext"
#include "kms.dcl"
#include "flags.def"


int serror;
int in, found;
int curr_op;
int dml_req_len;
int enum_length;        /* length of enumeration string */
int funct_count=0;
int use_prev_name;      /* boolean flag to indicate whether prev_name */
                        /* is used in function_type_declaration      */
int rel_operator;
int in_op;
int literal_type;

char temp_str[NUMDIGIT + 1];
char db[DBNLength + 1];
char temp_value[ENLength + 1];
char temp_name_id[ENLength + 1];
char temp[ENLength + 1];
char base_name[ENLength + 1];
char *enum_str;
char *var_str_alloc();
char *mem_ptr;

struct fun_dbid_node    *db_ptr;
struct ent_non_node     *ent_non_node_alloc(),
                        *ennptr;
struct ent_value*ent_value_alloc(),
                        *entval_ptr,
                        *entval_ptr2;
struct sub_non_node     *sub_non_node_alloc(),
                        *snnptr;
struct der_non_node     *der_non_node_alloc(),
                        *dnnptr;
```

```
struct ent_node          *ent_node_alloc(),
                         *search_entity(),
                         *entptr,
                         *last_entptr,
                         *new_entptr;
struct gen_sub_node      *new_gen_sub_node(),
                         *search_gensub(),
                         *subptr,
                         *new_subptr,
                         *last_subptr,
                         *the_subptr;
struct ent_node_list     *ent_node_list_alloc(),
                         *enl_ptr;
struct sub_node_list     *sub_node_list_alloc(),
                         *snl_ptr;
struct function_node     *function_node_alloc(),
                         *search_funct(),
                         *funct_ptr,
                         *last_funct_ptr,
                         *first_funct_ptr;
struct overlap_node      *overlap_node_alloc(),
                         *ovr_ptr;
struct dap_kms_info      *dap_kms_info_alloc();
struct ident_list *ident_list_alloc(),
                         *id_ptr,
                         *temp_ptr,
                         *new_temp_ptr;
struct dap_create_list   *create_list1,
                         *create_list2,
                         *dap_create_list_alloc();
struct dap_create_list   *dcl_ptr;
struct dap_av_pair_list  *av_ptr;
struct req_line_list     *req_ptr;

struct dml_statement     *dml_statement_alloc();
struct dap_expr_info     *dap_expr_info_alloc(),
                         *dap_expr_ptr;
struct relation_list     *relation_list_alloc(),
                         *rel_list_ptr;
struct funct_appln       *funct_appln_alloc(),
                         *funct_appln_ptr;
struct ent_info          *ent_info_ptr_alloc(),
                         *ent_info_ptr;
struct set_constructor   *set_constructor_alloc(),
                         *set_construct_ptr;
struct set_construct2    *set_construct2_alloc(),
                         *set_construct2_ptr;
struct set_construct3    *set_construct3_alloc(),
                         *set_construct3_ptr;
struct loop_info *loop_info_alloc(),
                         *loop_info_ptr;
```

```
struct domain_info      *domain_ptr;
struct simple_expr1     *simple_expr1_alloc(),
                        *simple_expr_ptr;
struct simple_expr2     *simple_expr2_alloc();
struct simple_expr3     *simple_expr3_alloc();
struct simple_expr4     *simple_expr4_alloc();
struct dap_range_info   *dap_range_info_alloc(),
                        *dap_range_ptr;
struct basic_expr_list  *basic_expr_list_alloc(),
                        *new_basic_expr_ptr,
                        *basic_expr_ptr;
struct order_comp_list  *order_comp_list_alloc(),
                        *order_comp_ptr;
struct comp_assoc_list  *comp_assoc_list_alloc(),
                        *new_comp_assoc_ptr,
                        *comp_assoc_ptr;
struct indexed_component        *indexed_component_alloc(),
                                *new_indexed_comp_ptr,
                                *indexed_comp_ptr;
struct dml_statement2_list      *dml_statement2_list_alloc(),
                                *dml_statement2_list_ptr;


%}

%union
     {
      char str[90];
     }


%token AND
%token ASCENDING
%token AVG
%token BOOLEAN
%token BY
%token CONSTANT
%token COUNT
%token CREATE
%token DATABASE
%token DELTA
%token DESCENDING
%token DESTROY
%token DIGITS
%token EACH
%token ELSE
%token END
%token ENTITY
%token EVERY
%token EXCLUDE
%token FALSE_T
```

```
%token FLOAT
%token FOR
%token FROM
%token IMAGE
%token INCLUDE
%token INTEGER
%token INTO
%token IN
%token IS
%token LOOP
%token MAX
%token MIN
%token MOVE
%token NEW
%token NOT
%token NO
%token NULL_T
%token OF
%token OR
%token OVERLAP
%token POS
%token PRINT_LINE
%token PRINT
%token RANGE
%token SET
%token SOME
%token STRING
%token SUBTYPE
%token SUM
%token THEN
%token TRUE_T
%token TYPE
%token UNIQUE
%token VALUE
%token VAL
%token WHERE
%token WITHIN
%token WITHNULL
%token WITHOUTNULL
%token WITH
%token XOR

%token <str> IDENTIFIER
%token <str> CHARACTER_STRING
%token <str> FLOAT_LITERAL
%token <str> INTEGER_LITERAL

%token ELIPSES
%token DOT
%token ASSIGN
%token COLON
```

```
%token IMPLY
%token EQ
%token NE
%token GE
%token GT
%token LE
%token LT
%token COMMA
%token SEMICOLON
%token LP
%token RP
%token LCB
%token RCB
%token HYPHEN


%start statement

%%

statement:
 ddl_statement
 | {
   db_ptr = cuser_dap_ptr->ui_li_type.li_dap.dpi_curr_db.cdi_db.dn_fun;
   }
 dml_statement
 ;


/* Three ident_list struct are maintained in dap_kms_info:     */
/*   1. Whenever a list of identifiers is read, the name_id of */
/*      each identifier is kept in the linked list dki_temp_ptr */
/*   2. When there is more than one list to be compared for    */
/*      uniqueness, the name_id of all identifiers are kept in */
/*      the link list dki_name1_ptr and dki_temp_ptr.          */
/*   3. The name_id of type or subtype declaration are kept in */
/*      the link list dki_id_ptr and dki_temp_ptr.             */

/* The new_id rule below will detect duplicate declaration of     */
/* name_id against dki_id_ptr, but will not issue an error        */
/* message. It will set the flag "found" on instead.              */
/* The id_list rule check duplication against dki_temp_ptr and    */
/* issue error message using the variable "serror". So,"serror"   */
/* must be set before using the rule id_list.                     */
/* Similarly, the name1 and name1_list rules use "serror" when    */
/* duplication is detected against dki_name1_ptr. So "serror"     */
/* must also be set before hand.                                  */

name_id: IDENTIFIER
   /* this rule assigns IDENTIFIER to the variable temp_value */
   /* and trancates it if it is longer then ENLength          */
```

```
   {
#ifdef DYacFlag
        printf("name_id %s recognized\n",$1);
#endif
    strncpy(temp_value,$1,ENLength);
    if (strlen($1) > ENLength)
        printf("warning: name_id %s trancated to %s\n",$1,temp_value);
   }
  ;

new_id: name_id
  /* Check for duplicate declaration against list pointed to */
  /* by dki_id_ptr. If not found, insert it to the list for  */
  /* subsequent comparisons of uniqueness                    */
   {
#ifdef DYacFlag
        printf("new_id recognized\n");
#endif
    found = FALSE;
    id_ptr = kms_ptr->dki_id_ptr;
    while(id_ptr != NULL && (!found))
     {
        if (strcmp(id_ptr->name, temp_value) == FALSE)
          found = TRUE;
        else
          id_ptr = id_ptr->next;
     }   /* end while */
    if (!found)
     {
        id_ptr = ident_list_alloc();
        strcpy(id_ptr->name, temp_value);
        id_ptr->next = kms_ptr->dki_id_ptr;
        kms_ptr->dki_id_ptr = id_ptr;
     }
   }
 ;

new_id_list:     /* use dki_temp_ptr to keep list of name_id */
  new_id
   {
    if (found)
         proc_eval_error(serror,temp_value);
    free_temp_list();
    kms_ptr->dki_temp_ptr = ident_list_alloc();
    strcpy(kms_ptr->dki_temp_ptr->name, temp_value);
    kms_ptr->dki_temp_ptr->next = NULL;
   }
  | new_id_list COMMA new_id
   {
    if (found)
         proc_eval_error(serror,temp_value);
```

```
    temp_ptr = ident_list_alloc();
    strcpy(temp_ptr->name, temp_value);
    temp_ptr->next = kms_ptr->dki_temp_ptr;
    kms_ptr->dki_temp_ptr = temp_ptr;
   }
  ;

id_list:
  name_id
   {
    free_temp_list();
    kms_ptr->dki_temp_ptr = ident_list_alloc();
    strcpy(kms_ptr->dki_temp_ptr->name, temp_value);
    kms_ptr->dki_temp_ptr->next = NULL;
   }
  | id_list COMMA name_id
   {
    found = FALSE;
    temp_ptr = kms_ptr->dki_temp_ptr;
    while ((temp_ptr != NULL) && (!found))
        {
         if (strcmp(temp_ptr->name, temp_value)== FALSE)
           found = TRUE;
         else
           temp_ptr = temp_ptr->next;
        }
    if (!found)
        {
         temp_ptr = ident_list_alloc();
         strcpy(temp_ptr->name, temp_value);
         temp_ptr->next = kms_ptr->dki_temp_ptr;
         kms_ptr->dki_temp_ptr = temp_ptr;
        }
    else   /* found duplicate */
         proc_eval_error(serror,temp_value);
   }
  ;

name1: name_id
   {
#ifdef DYacFlag
        printf("enter name1\n");
#endif
    found = FALSE;
    temp_ptr = kms_ptr->dki_name1_ptr;
    while ((temp_ptr != NULL) && (!found))
      {
         if (strcmp(temp_ptr->name, temp_value)== FALSE)
           found = TRUE;
       else
           temp_ptr = temp_ptr->next;
```

```
      }
    if (!found)
      {
         id_ptr = ident_list_alloc();
         strcpy(id_ptr->name, temp_value);
         id_ptr->next = kms_ptr->dki_name1_ptr;
         kms_ptr->dki_name1_ptr = id_ptr;
      }
    else
         proc_eval_error(serror,temp_value);
#ifdef DYacFlag
         printf("exit name1 proc\n");
#endif
    }
 ;

name1_list:  /* remember to call free_name1_list before using */
            /* name1_list for the first list */
 name1
         {
          free_temp_list();
          kms_ptr->dki_temp_ptr = ident_list_alloc();
          strcpy(kms_ptr->dki_temp_ptr->name, temp_value);
          kms_ptr->dki_temp_ptr->next = NULL;
         }
 | name1_list COMMA name1
         {
          temp_ptr = ident_list_alloc();
          strcpy(temp_ptr->name, temp_value);
          temp_ptr->next = kms_ptr->dki_temp_ptr;
          kms_ptr->dki_temp_ptr = temp_ptr;
         }
 ;

         /*********************/
         /**  ddl statement  **/
         /*********************/

ddl_statement:
 DATABASE name_id         /* database name */

 /*  the fun_dbid_node is located and compared for correctness  */

   {
#ifdef DYacFlag
         printf("name_id in ddl_statement recognized\n");
#endif
    db_ptr = cuser_dap_ptr->ui_li_type.li_dap.dpi_curr_db.cdi_db.dn_fun;
    if (strcmp(temp_value, db_ptr->fdn_name) != FALSE)
         {
          serror = 0;
```

```
            proc_eval_error(serror,temp_value);
            YYACCEPT;
           }
      kms_ptr->dki_id_ptr = ident_list_alloc();
      strcpy(kms_ptr->dki_id_ptr->name, temp_value);
      kms_ptr->dki_id_ptr->next = NULL;
     }
   IS declarative_item_list end_database SEMICOLON
   ;

end_database:
   END
  | END name_id
     {
      if (strcmp(temp_value, db_ptr->fdn_name) != FALSE)
           printf("warning: different db name declared at end, ignored\n");
     }
   ;

declarative_item_list: declarative_item
                     | declarative_item_list declarative_item
                     ;

declarative_item: declaration
                | overlap_rule
                | uniqueness_rule
                     | error SEMICOLON
                ;

overlap_rule:
   OVERLAP
     {
      serror = 13; /* duplicate identifier in list */
      free_name1_list();
     }
   name1_list
     {
#ifdef DYacFlag
      printf("first name1_list in overlap_rule part recognized\n");
#endif
      ovr_ptr = overlap_node_alloc();
      ovr_ptr->num_sub_node = 0;
      ovr_ptr->snlptr = NULL;
      temp_ptr = kms_ptr->dki_temp_ptr;
      while (temp_ptr != NULL)
           {
            subptr = db_ptr->fdn_subptr;
            in = FALSE;
            while ((subptr != NULL) && (!in))
              {
               if (strcmp(temp_ptr->name, subptr->gsn_name) == FALSE)
```

```
            {
              in = TRUE;
              /* overlap types must be terminal subtype */
              if (!(subptr->gsn_terminal))
                proc_eval_error(14,temp_ptr->name);
              else
               {
                find_base_type(subptr,base_name);
                if (ovr_ptr->num_sub_node == 0)
                      strcpy(ovr_ptr->base_type_name,base_name);
                else if (strcmp(ovr_ptr->base_type_name,base_name))
                      /* overlap types must have same base type */
                      proc_eval_error(20,temp_ptr->name);
                snl_ptr = sub_node_list_alloc();
                snl_ptr->subptr = subptr;
                snl_ptr->next = ovr_ptr->snlptr;
                ovr_ptr->snlptr = snl_ptr;
                ovr_ptr->num_sub_node++;
               }
            }
           else
                subptr = subptr->gsn_next_genptr;
          }  /* end while */
         if (!in)  /* undeclared subtype name */
           proc_eval_error(10,temp_ptr->name);
         temp_ptr = temp_ptr->next;
        }
   }
  WITH name1_list SEMICOLON
   {
#ifdef DYacFlag
    printf("second name1_list in overlap_rule part recognized\n");
#endif
    temp_ptr = kms_ptr->dki_temp_ptr;
    while (temp_ptr != NULL)
     {
       subptr = db_ptr->fdn_subptr;
       in = FALSE;
       while ((subptr != NULL) && (!in))
        {
         if (strcmp(subptr->gsn_name, temp_ptr->name) == FALSE)
           {
              in = TRUE;
              /* overlap types must be terminal subtype */
              if (!(subptr->gsn_terminal))
                proc_eval_error(14,temp_ptr->name);
              else
               {
                find_base_type(subptr,base_name);
                if (strcmp(ovr_ptr->base_type_name,base_name))
                      /* overlap types must have same base type */
```

```
                    proc_eval_error(20,temp_ptr->name);
                snl_ptr = sub_node_list_alloc();
                snl_ptr->subptr = subptr;
                snl_ptr->next = ovr_ptr->snlptr;
                ovr_ptr->snlptr = snl_ptr;
                ovr_ptr->num_sub_node++;
                }
            }    /* end in = TRUE */

          else   /* strcmp == TRUE */
                subptr = subptr->gsn_next_genptr;
          }

        if (!in)  /* undeclared subtype name */
          proc_eval_error(10,temp_ptr->name);

        temp_ptr = temp_ptr->next;

      } /* end while temp_ptr */
    ovr_ptr->next = db_ptr->fdn_ovrptr;
    db_ptr->fdn_ovrptr = ovr_ptr;
    db_ptr->fdn_num_ovr++;
#ifdef DYacFlag
        printf("overlap_rule part done\n");
#endif
   }
 ;

uniqueness_rule:
 UNIQUE
  {
#ifdef DYacFlag
    printf("uniqueness_rule part recognized\n");
#endif
    serror = 13;    /* duplicate identifier in list */
  }
 id_list WITHIN name_id SEMICOLON
  {
        entptr = db_ptr->fdn_entptr;
        subptr = db_ptr->fdn_subptr;
        in = FALSE;
        while ((entptr != NULL) && (!in))
         {
          if (strcmp(entptr->en_name, temp_value) == FALSE)
            {
                in = TRUE;
                first_funct_ptr = entptr->en_ftnptr;
            }
          else
                entptr = entptr->en_next_ent;
         } /* end while entptr */
```

```
        while ((subptr != NULL) && (!in))
         {
          if (strcmp(subptr->gsn_name, temp_value) == FALSE)
            {
                in = TRUE;
                first_funct_ptr = subptr->gsn_ftnptr;
            }
          else
                subptr = subptr->gsn_next_genptr;
           }  /* end while subptr */

        if (!in)      /* undefined entity type or subtype */
          proc_eval_error(15,temp_value);
        else
         /* find the function node and set its unique field to true */
         {
          temp_ptr = kms_ptr->dki_temp_ptr;
          while (temp_ptr != NULL)
           {
            funct_ptr = first_funct_ptr;
            found = FALSE;
            while ((funct_ptr != NULL) && (!found))
             {
                if(strcmp(temp_ptr->name, funct_ptr->fn_name) == FALSE)
                 {
                  found = TRUE;
                  funct_ptr->fn_unique = TRUE;
                 }
                else
                  funct_ptr = funct_ptr->next;
             }  /* end while funct_ptr */

            if (!found)  /* undelared function identifier */
                proc_eval_error(6,temp_ptr->name);

            temp_ptr = temp_ptr->next;
           }  /* end while temp_ptr */
         }
#ifdef DYacFlag
        printf("uniqueness_rule part done\n");
#endif
        }
  ;

declaration: number_declaration
        | TYPE new_id
              {
#ifdef DYacFlag
    printf("type_declaration recognized\n");
#endif
                strcpy(temp_name_id, temp_value);
```

```
            }
          type_declaration
            {
#ifdef DYacFlag
        printf("type_declaration done\n");
#endif
            }
        | SUBTYPE new_id
            {
#ifdef DYacFlag
    printf("subtype_declaration recognized\n");
#endif
                strcpy(temp_name_id, temp_value);
            }
          subtype_declaration
            {
#ifdef DYacFlag
        printf("subtype_declaration done\n");
#endif
            }
      ;

number_declaration:      /* section 4.2.5 */
    {
      serror = 1;        /* duplicate constant name */
    }
 new_id_list COLON CONSTANT ASSIGN simple_const SEMICOLON
    {
      temp_ptr = kms_ptr->dki_temp_ptr;
      while (temp_ptr != NULL)
       {
        /* At this point ent_non_node's are filled with the */
        /* information previously allocated in the kms info */
        /* structure.  The amount of nodes is dependent on  */
        /* the amount of names in the temp structure.       */

        add_ent_non_node(temp_ptr->name);
        temp_ptr = temp_ptr->next;
       }
    }
 ;

simple_const:
      /* dap_kms_info structures are built for subsequent */
      /* nonentity node insertion into the schema         */

 INTEGER_LITERAL
      {
       put_dki_ent_non('i', $1);
      }
 | FLOAT_LITERAL
```

```
        {
          put_dki_ent_non('f', $1);
        }
  | CHARACTER_STRING
        {
          put_dki_ent_non('s', $1);
        }
  ;

type_declaration:
  IS type_definition SEMICOLON

    /* the following switch statement allocates a nonentity or derived */
    /* node to the schema dependent upon the value of curr_op          */

    {
     switch(curr_op)
       {
          case NonEnt:
            add_ent_non_node(temp_name_id);
            break;

          case Derived:
            dnnptr = der_non_node_alloc();
            strcpy(dnnptr->dnn_name, temp_name_id );
            dnnptr->dnn_type = kms_ptr->dki_ent_non.enn_type ;
            dnnptr->dnn_total_length =
                kms_ptr->dki_ent_non.enn_total_length ;
            dnnptr->dnn_range = kms_ptr->dki_ent_non.enn_range ;
            dnnptr->dnn_num_values =
                kms_ptr->dki_ent_non.enn_num_values ;
            dnnptr->dnn_value = kms_ptr->dki_ent_non.enn_value ;
            kms_ptr->dki_ent_non.enn_value = NULL;
            dnnptr->dnn_next_node = db_ptr->fdn_nonderptr;
            db_ptr->fdn_nonderptr = dnnptr;
            break;

       } /* end switch */

    }
  | IS entity_type_definition SEMICOLON
    {
          /* check if name_id is on the ent list of the schema */
      entptr = db_ptr->fdn_entptr;
      in = FALSE;
      while ((entptr != NULL) && (!in))
          {
           if (strcmp(entptr->en_name, temp_name_id) == FALSE)
             in = TRUE;
           else
             entptr = entptr->en_next_ent;
```

```
        }
    if (!in)
        {
         entptr = ent_node_alloc();
         strcpy(entptr->en_name, temp_name_id);
         entptr->en_last_ent_id = 0;
         entptr->en_num_funct = 0;
         entptr->en_terminal = TRUE;
         entptr->en_ftnptr = NULL;
         entptr->en_next_ent = db_ptr->fdn_entptr;
         db_ptr->fdn_entptr = entptr;
         db_ptr->fdn_num_ent++;
        }
    if (entptr->en_num_funct == 0)
        {
         entptr->en_ftnptr = funct_ptr;
         entptr->en_num_funct = entptr->en_num_funct + funct_count;
        }
    else
         proc_eval_error(3,temp_name_id);   /* duplicate entity type name */
    funct_count = 0;
    last_funct_ptr = NULL;
   }
 | incomplete_type_declaration
 ;

incomplete_type_declaration:    /* entity */
  SEMICOLON
   {
#ifdef DYacFlag
    printf("incomplete_type_declaration recognized\n");
#endif
    if (found)     /* duplicate entity type */
        proc_eval_error(3,temp_value);
    else
       {
    /* add entity node here */
        entptr = ent_node_alloc();
        strcpy(entptr->en_name, temp_value);
        entptr->en_last_ent_id = 0;
        entptr->en_num_funct = 0;
        entptr->en_terminal = TRUE;
        entptr->en_ftnptr = NULL;
        entptr->en_next_ent = db_ptr->fdn_entptr;
        db_ptr->fdn_entptr = entptr;
        db_ptr->fdn_num_ent++;
       }
#ifdef DYacFlag
    printf("incomplete_type_declaration done\n");
#endif
   }
```

```
 ;

type_definition:
 enumeration_type_definition
        { curr_op = NonEnt; }
 | integer_range
        { curr_op = NonEnt; }
 | float_range
        { curr_op = NonEnt; }
 | derived_type_definition
        { curr_op = Derived; }
 ;

enumeration_type_definition:
 LP
   /* enumeration dap_kms_info structures for */
   /* nonentity and function nodes are initialized */

   {
#ifdef DYacFlag
        printf("LP in database_specification recognized\n");
#endif
        kms_ptr->dki_ent_non.enn_type = 'e';
        kms_ptr->dki_ent_non.enn_range = TRUE;
        kms_ptr->dki_ent_non.enn_num_values = 0;
        kms_ptr->dki_ent_non.enn_constant = FALSE;
   }
 enumeration_literal_list RP
 ;

enumeration_literal_list:
 enumeration_literal

          /* the pointers are set for value nodes with */
          /* concurrent incrementation of the number of */
          /* value nodes present in the nonentity and   */
          /* function structures                        */

    {
        kms_ptr->dki_ent_non.enn_num_values++;
        kms_ptr->dki_ent_non.enn_value = entval_ptr;
        kms_ptr->dki_ent_non.enn_total_length = enum_length;
        entval_ptr = NULL;
    }

 | enumeration_literal_list COMMA enumeration_literal

    {
        kms_ptr->dki_ent_non.enn_num_values++;
        entval_ptr2 = kms_ptr->dki_ent_non.enn_value;
        while (entval_ptr2->next != NULL)
```

```
            entval_ptr2 = entval_ptr2->next;
        entval_ptr2->next = entval_ptr;
      if (enum_length > kms_ptr->dki_ent_non.enn_total_length)
          kms_ptr->dki_ent_non.enn_total_length = enum_length;
        entval_ptr = NULL;
    }
 ;


enumeration_literal: IDENTIFIER
    {
#ifdef DYacFlag
        printf("enumeration_literal recognized\n");
#endif
        entval_ptr = ent_value_alloc();
        enum_length = strlen($1) + 1;
        entval_ptr->ev_value = var_str_alloc(enum_length);
        strcpy(entval_ptr->ev_value, $1);
        entval_ptr->next = NULL;
    }
 ;

integer_range: RANGE int_range
 ;

int_range :INTEGER_LITERAL ELIPSES INTEGER_LITERAL
    {
        add_range_non_node('i', $1, $3);
    }
 ;

float_range: RANGE FLOAT_LITERAL ELIPSES FLOAT_LITERAL
    {
        add_range_non_node('f', $2, $4);
    }
 ;

derived_type_definition: NEW name_id derived_range

     /* the nonentity, subtype nonentity, and derived type */
     /* nonentity nodes are examined to find which con-    */
     /* the current value of IDENTIFIER                    */

    {
        serror = 16;      /* incompatable derived type */
        compare_non_node_type(temp_value, serror);
    }
 ;

derived_range:            /* enumeration type not included? section 4.2.4 */
 integer_range
```

```
 | float_range
 ;

entity_type_definition:
 ENTITY
  {
   free_name1_list();
  }
 entity_component_declaration_list END ENTITY
 | ENTITY END ENTITY
 ;

entity_component_declaration_list:
 entity_component_declaration
 | entity_component_declaration_list entity_component_declaration
 ;

entity_component_declaration:
 name1_list COLON
  {
   /* initialize use_prev_name flag */
   use_prev_name = FALSE;
  }
 function_type_declaration
  {
#ifdef DYacFlag
    printf("function_type_declaration in entity_compo_declaration\n");
#endif
    temp_ptr = kms_ptr->dki_temp_ptr;
    while (temp_ptr != NULL)
     {
        funct_count++;
        funct_ptr = function_node_alloc();
        strcpy(funct_ptr->fn_name, temp_ptr->name);
        if (use_prev_name)
          funct_ptr->fn_type = 'e';
        else
          funct_ptr->fn_type = kms_ptr->dki_funct.fn_type;
        funct_ptr->fn_range = kms_ptr->dki_funct.fn_range;
        funct_ptr->fn_total_length = kms_ptr->dki_funct.fn_total_length;
        funct_ptr->fn_num_value = kms_ptr->dki_funct.fn_num_value;
        funct_ptr->fn_value =  kms_ptr->dki_funct.fn_value;
        funct_ptr->fn_entptr = kms_ptr->dki_funct.fn_entptr;
        funct_ptr->fn_subptr = kms_ptr->dki_funct.fn_subptr;
        funct_ptr->fn_nonentptr = kms_ptr->dki_funct.fn_nonentptr;
        funct_ptr->fn_nonsubptr = kms_ptr->dki_funct.fn_nonsubptr;
        funct_ptr->fn_nonderptr = kms_ptr->dki_funct.fn_nonderptr;
        funct_ptr->fn_entnull = kms_ptr->dki_funct.fn_entnull;
        funct_ptr->fn_unique = kms_ptr->dki_funct.fn_unique;
        funct_ptr->next = last_funct_ptr;
        last_funct_ptr = funct_ptr;
```

```
        temp_ptr = temp_ptr->next;
       }
   }
   error SEMICOLON
  ;

function_type_declaration:
  function_type end_scalar_function
 | set_type_definition SEMICOLON
  ;

set_type_definition: SET OF function_type
   {
        kms_ptr->dki_funct.fn_set = TRUE;
   }
  ;

end_scalar_function:
  SEMICOLON
 | ASSIGN
   {
    serror = 18;   /* incompatible assignment */
    kms_ptr->dki_funct.fn_num_value = 1;
    kms_ptr->dki_funct.fn_value = ent_value_alloc();
    kms_ptr->dki_funct.fn_value->next = NULL;
   }
  default_value SEMICOLON
  ;

default_value:
  INTEGER_LITERAL
   {
    if (kms_ptr->dki_funct.fn_type != 'i')
        proc_eval_error(serror,$1);    /* incompatible assignment */
    kms_ptr->dki_funct.fn_value->ev_value = var_str_alloc(strlen($1)+1);
    strcpy(kms_ptr->dki_funct.fn_value->ev_value,$1);
   }
 | FLOAT_LITERAL
   {
    if (kms_ptr->dki_funct.fn_type != 'f')
        proc_eval_error(serror,$1);    /* incompatible assignment */
    kms_ptr->dki_funct.fn_value->ev_value = var_str_alloc(strlen($1)+1);
    strcpy(kms_ptr->dki_funct.fn_value->ev_value,$1);
   }
 | CHARACTER_STRING
   {
    if (kms_ptr->dki_funct.fn_type != 's')
        proc_eval_error(serror,$1);    /* incompatible assignment */
    kms_ptr->dki_funct.fn_value->ev_value = var_str_alloc(strlen($1)+1);
    strcpy(kms_ptr->dki_funct.fn_value->ev_value,$1);
   }
```

```
  | boolean_value
    {
     if (kms_ptr->dki_funct.fn_type != 'b')
          proc_eval_error(serror,"");    /* incompatible assignment */
    }
  ;

boolean_value:
  TRUE_T
    {
     kms_ptr->dki_funct.fn_value->ev_value = var_str_alloc(2);
     strcpy(kms_ptr->dki_funct.fn_value->ev_value,"1");
    }
  | FALSE_T
    {
     kms_ptr->dki_funct.fn_value->ev_value = var_str_alloc(2);
     strcpy(kms_ptr->dki_funct.fn_value->ev_value,"0");
    }
  ;

function_type:
  type_mark constraint
  | STRING LP string_range RP
    {
     put_dki_funct('s');
     kms_ptr->dki_funct.fn_total_length =
       str_to_num(kms_ptr->dki_ent_non.enn_value->next->ev_value) -
       str_to_num(kms_ptr->dki_ent_non.enn_value->ev_value) + 1;
    }
  ;

string_range:
  int_range
  | INTEGER_LITERAL
    {
     add_range_non_node('i', "1", $1);
    }
  ;

type_mark:
  name_id
    {
          use_prev_name = TRUE;
          ennptr = db_ptr->fdn_nonentptr;
          entptr = db_ptr->fdn_entptr;
          subptr = db_ptr->fdn_subptr;
          snnptr = db_ptr->fdn_nonsubptr;
          dnnptr = db_ptr->fdn_nonderptr;
          in = FALSE;
          while ((ennptr != NULL) && (!in))
            {
```

```
    if (strcmp(ennptr->enn_name, temp_value) == FALSE)
      {
          in = TRUE;
          put_dki_funct(ennptr->enn_type);
          kms_ptr->dki_funct.fn_total_length =
                          ennptr->enn_total_length;
          kms_ptr->dki_funct.fn_nonentptr = ennptr;
      }
    else
          ennptr = ennptr->enn_next_node;
  }

while ((snnptr != NULL) && (!in))
  {
    if (strcmp(snnptr->snn_name, temp_value) == FALSE)
      {
          in = TRUE;
          put_dki_funct(snnptr->snn_type);
          kms_ptr->dki_funct.fn_total_length =
                          snnptr->snn_total_length;
          kms_ptr->dki_funct.fn_nonsubptr = snnptr;
      }
    else
          snnptr = snnptr->snn_next_node;
  }

while ((dnnptr != NULL) && (!in))
  {
    if (strcmp(dnnptr->dnn_name, temp_value) == FALSE)
      {
          in = TRUE;
          put_dki_funct(dnnptr->dnn_type);
          kms_ptr->dki_funct.fn_total_length =
                          dnnptr->dnn_total_length;
          kms_ptr->dki_funct.fn_nonderptr = dnnptr;
      }
    else
          dnnptr = dnnptr->dnn_next_node;
  }

while ((subptr != NULL) && (!in))
  {
    if (strcmp(subptr->gsn_name, temp_value) == FALSE)
      {
          in = TRUE;
          put_dki_funct('E');
          kms_ptr->dki_funct.fn_subptr = subptr;
      }
    else
          subptr = subptr->gsn_next_genptr;
  }
```

```
        while ((entptr != NULL) && (!in))
          {
           if (strcmp(entptr->en_name, temp_value) == FALSE)
             {
                in = TRUE;
                put_dki_funct('E');
                kms_ptr->dki_funct.fn_entptr = entptr;
             }
           else
                entptr = entptr->en_next_ent;
          }

        if (in)
          {
           strcpy(temp, temp_value);
          }
        else   /* id must be previously declared */
           proc_eval_error(19,temp_value);
  }
  | FLOAT
    {
        put_dki_funct('f');
    }
  | INTEGER
    {
        put_dki_funct('i');
    }
  | BOOLEAN
    {
        put_dki_funct('b');
    }
  ;

constraint:
    /* empty */
  | range_constraint
  | null_constraint
    {
    if (kms_ptr->dki_funct.fn_type != 'E')
      {
       printf("warning: illegal null constraint declaration ignored\n");
       kms_ptr->dki_funct.fn_entnull = FALSE;
      }
    }
  ;

range_constraint:
  /* range constraints here are ignored, no action necessary */
  integer_range
  | float_range
  | enumeration_range
```

```
 ;

enumeration_range: RANGE IDENTIFIER ELIPSES IDENTIFIER
 ;

null_constraint:
 WITHNULL
  {
   kms_ptr->dki_funct.fn_entnull = TRUE;
  }
 | WITHOUTNULL
 ;

subtype_declaration:
 IS
  {
   serror = 13;     /* duplicate id in list */
  }
 complete_subtype SEMICOLON
 | incomplete_subtype_declaration
 ;

complete_subtype:
 name_id subtype_definition
   {
        serror = 17;       /* incompatable subtype declaration */
        compare_non_node_type(temp_value, serror);

        /* create sub_non_node */
        snnptr = sub_non_node_alloc();
        strcpy(snnptr->snn_name, temp_value);
        snnptr->snn_type = kms_ptr->dki_ent_non.enn_type;
        snnptr->snn_total_length = kms_ptr->dki_ent_non.enn_total_length;
        snnptr->snn_range = kms_ptr->dki_ent_non.enn_range;
        snnptr->snn_num_values = kms_ptr->dki_ent_non.enn_num_values;
        snnptr->snn_value = kms_ptr->dki_ent_non.enn_value;
        kms_ptr->dki_ent_non.enn_value = NULL;
        snnptr->snn_next_node = db_ptr->fdn_nonsubptr;
        db_ptr->fdn_nonsubptr = snnptr;
   }
 | id_list
   {
    /* locate or create the gen sub node */
    the_subptr = db_ptr->fdn_subptr;
    in = FALSE;
    while ((the_subptr != NULL) && (!in))
     {
        if (strcmp(the_subptr->gsn_name, temp_name_id) == FALSE)
          in = TRUE;
        else
          the_subptr = the_subptr->gsn_next_genptr;
```

```
  }
if (!in)
     the_subptr = new_gen_sub_node(temp_name_id);

/* For each name_id in id_list,locate the gen_sub_node   */
/* or the ent_node and change them to non_terminal type. */
/* Corresponding overlap node is created and the_subptr  */
/* is updated.                                           */

temp_ptr = kms_ptr->dki_temp_ptr;
while (temp_ptr != NULL)
  {
   subptr = db_ptr->fdn_subptr:
     entptr = db_ptr->fdn_entptr:
   in = FALSE:
     while ((subptr != NULL) && (!in))
       {
        if (strcmp(subptr->gsn_name, temp_ptr->name) == FALSE)
         {
          in = TRUE:
          subptr->gsn_terminal = FALSE:
          the_subptr->gsn_num_sub--:
          snl_ptr = sub_node_list_alloc();
          snl_ptr->subptr = subptr:
          snl_ptr->next = the_subptr->gsn_subptr;
          the_subptr->gsn_subptr = snl_ptr;
         }
        else
          subptr = subptr->gsn_next_genptr;
       }

     while ((entptr != NULL) && (!in))
       {
        if (strcmp(entptr->en_name, temp_ptr->name) == FALSE)
         {
          in = TRUE;
          entptr->en_terminal = FALSE;
          the_subptr->gsn_num_ent++;
          enl_ptr = ent_node_list_alloc();
          enl_ptr->entptr = entptr;
          enl_ptr->next = the_subptr->gsn_entptr;
          the_subptr->gsn_entptr = enl_ptr;
         }
        else
          entptr = entptr->en_next_ent;
       }
     if (!in)       /* use of undefined entity type or subtype */
       proc_eval_error(15,temp_ptr->name);
     temp_ptr = temp_ptr->next;
  }
}
```

```
entity_type_definition
  {
   if (the_subptr->gsn_num_funct ==0)
     {
        the_subptr->gsn_ftnptr = funct_ptr:
        the_subptr->gsn_num_funct = funct_count;
     }
   else     /* entity subtype name must be unique */
        proc_eval_error(12,temp_name_id);
   funct_count = 0:
   last_funct_ptr = NULL;
  }
| STRING LP string_range RP
;


subtype_definition:
 RANGE enumeration_literal ELIPSES enumeration_literal
 | integer_range
 | float_range
 | /* empty */
 ;

incomplete_subtype_declaration:
 SEMICOLON /* entity */
  {
   if (!found)
        subptr = new_gen_sub_node(temp_value);
   else  /* duplicate subtype */
        proc_eval_error(12,temp_value);
  }
 ;

        /*********************/
        /**  dml statement  **/
        /*********************/

dml_statement:
   create_statement
 | destroy_statement
 | move_statement
 | loop_statement
 ;

dml_statement2:
   assignment_statement
 | include_statement
 | exclude_statement
 | destroy_statement
 | move_statement
 | procedure_call
```

```
;

create_statement:
 CREATE NEW
  {
   kms_ptr->dml_statement_ptr = dml_statement_alloc();
   kms_ptr->dml_statement_ptr->type = Create;
   kms_ptr->dml_statement_ptr->dap_expr_ptr = NULL;
   kms_ptr->dml_statement_ptr->indexed_comp_ptr = NULL;
   kms_ptr->dml_statement_ptr->basic_expr_ptr = NULL;
   kms_ptr->dml_statement_ptr->comp_assoc_ptr = NULL:
  }
 create_part SEMICOLON
  {
   kms_ptr->dml_statement_ptr->comp_assoc_ptr = comp_assoc_ptr;
  }
 ;

create_part:
  {
   serror = 13:  /* duplicate identifiers in list */
  }
 id_list
  {
   /* perform error checking and fill structure dap_create_list */
   /* All functions for each name_id in the list is attached on */
   /* dap_createList pointed to by kms_ptr->dki_create  */
   if (!proc_create_ent_type(db_ptr->fdn_entptr))
        proc_create_sub_type(db_ptr->fdn_subptr);
  }
 named_aggregate
  {
   build_req_line_list():
   /* temp for program testing */
   req_ptr = kms_ptr->dki_req_ptr;
   while (req_ptr != NULL)
        {
         printf("%s\n",req_ptr->req_line);
         req_ptr = req_ptr->next;
        }
  }
 ;

named_aggregate:
 LP component_association_list RP
 | /* empty */
  {
   comp_assoc_ptr = NULL;
  }
 ;
```

```
component_association_list:
 component_association
  {
   kms_ptr->dml_statement_ptr->comp_assoc_ptr = new_comp_assoc_ptr;
   comp_assoc_ptr = kms_ptr->dml_statement_ptr->comp_assoc_ptr;
   comp_assoc_ptr->next = NULL;
  }
 | component_association_list COMMA component_association
  {
   new_comp_assoc_ptr->next = comp_assoc_ptr;
   comp_assoc_ptr = new_comp_assoc_ptr;
  }
 ;

component_association:
 IDENTIFIER
  {
   /* check if name_id is a valid attribute name */
   dcl_ptr = kms_ptr->dki_create;
   in = FALSE;
   while ((dcl_ptr != NULL) && (!in))
    {
       av_ptr = dcl_ptr->av_pair_ptr;
       while ((av_ptr != NULL) && (!in))
        {
         if (strcmp(av_ptr->ftnptr->fn_name, $1) == 0)
           in = TRUE;
         else
           av_ptr = av_ptr->next;
        }
       dcl_ptr = dcl_ptr->next;
    }
   if (!in)    /* invalid function name */
       proc_eval_error(26,$1);
   else
       {
        new_comp_assoc_ptr = comp_assoc_list_alloc();
        strcpy(new_comp_assoc_ptr->name, $1);
        simple_expr_ptr = &(new_comp_assoc_ptr->simple_expr);
       }
   clean_dki_evl_ptr();
  }
 IMPLY simple_expr
  {
   av_ptr->num_value = kms_ptr->dki_evl_ptr.num_values;
   av_ptr->valptr = kms_ptr->dki_evl_ptr.ev_ptr;
   kms_ptr->dki_evl_ptr.ev_ptr = NULL;
  }
 ;
```

```
literal :
 CHARACTER_STRING
  {
  literal_type = String;
  put_dki_ent_value_list(literal_type,$1);
  }
 | INTEGER_LITERAL
  {
  literal_type = Integer;
  put_dki_ent_value_list(literal_type,$1);
  }
 | FLOAT_LITERAL
  {
  literal_type = Float;
  put_dki_ent_value_list(literal_type,$1);
  }
 | IDENTIFIER    /* Can be Entity, GenSub or Enumeration constant */
  {
  if (search_entity($1) != NULL)
       literal_type = Entity;
  else if (search_gensub($1) != NULL)
       literal_type = GenSub;
  else
       literal_type = Enum;
  put_dki_ent_value_list(literal_type,$1);
  }
 | TRUE_T
  {
  literal_type = Boolean;
  put_dki_ent_value_list(literal_type,"1");
  }
 | FALSE_T
  {
  literal_type = Boolean;
  put_dki_ent_value_list(literal_type,"0");
  }
 ;

/*
  Memory must be allocated before activating the following rules:

   dap_expr                    uses dap_expr_ptr
   simple_expr          uses simple_expr_ptr
   dap_range                   uses dap_range_ptr
   domain                      uses domain_ptr

  Rules that allocate memory themselves :

   comp_assco_list             returns comp_assoc_ptr;
   relation                    returns rel_list_ptr
   basic_expr_list             returns basic_expr_ptr
```

```
    indexed_component          returns indexed_comp_ptr
    set_constructor            returns set_construct_ptr
    function_application returns funct_appln_ptr

*/

set_constructor:
  LCB
    {
     set_construct_ptr = set_constructor_alloc();
     simple_expr_ptr = simple_expr1_alloc();
    }
  end_set_constructor
  ;

end_set_constructor:
  basic_expr_list RCB
    {
     set_construct_ptr->basic_expr_ptr = basic_expr_ptr;
     basic_expr_ptr = NULL;
    }
   simple_expr IN set_constructor_part WHERE dap_expr RCB
   RCB      /* empty or null list */
    {
     literal_type = String;
     put_dki_ent_value_list(literal_type,"$NULL");
     set_construct_ptr->basic_expr_ptr = NULL;
     set_construct_ptr->set_construct2_ptr = NULL;
     set_construct_ptr->set_construct3_ptr = NULL;
    }
  ;

set_constructor_part:
    {
     set_construct2_ptr = set_construct2_alloc();
     set_construct_ptr->set_construct2_ptr = set_construct2_ptr;
     set_construct2_ptr->simple_expr1_ptr = simple_expr_ptr;
     dap_expr_ptr = &(set_construct2_ptr->dap_expr_ptr);
    }
  name1
  | {
     set_construct3_ptr = set_construct3_alloc();
     set_construct_ptr->set_construct3_ptr = set_construct3_ptr;
     set_construct3_ptr->simple_expr1_ptr = simple_expr_ptr;
     dap_range_ptr = &(set_construct3_ptr->dap_range);
     dap_expr_ptr = &(set_construct3_ptr->dap_expr_ptr);
    }
  dap_range
  ;

basic_expr :
```

```
 literal
   {
    new_basic_expr_ptr = basic_expr_list_alloc();
    strcpy(new_basic_expr_ptr->lit_array,kms_ptr->dki_evl_ptr.ev_ptr);
    new_basic_expr_ptr->indexed_comp_ptr = NULL;
    new_basic_expr_ptr->funct_appln_ptr = NULL;
   }
 | indexed_component
   {
    new_basic_expr_ptr = basic_expr_list_alloc();
    new_basic_expr_ptr->indexed_comp_ptr = indexed_comp_ptr;
    new_basic_expr_ptr->funct_appln_ptr = NULL;
    strcpy(new_basic_expr_ptr->lit_array,"");
    indexed_comp_ptr = NULL;
   }
 | function_application
   {
    new_basic_expr_ptr = basic_expr_list_alloc();
    new_basic_expr_ptr->funct_appln_ptr = funct_appln_ptr;
    new_basic_expr_ptr->indexed_comp_ptr = NULL;
    strcpy(new_basic_expr_ptr->lit_array,"");
    funct_appln_ptr = NULL;
   }
 ;

basic_expr_list :
 basic_expr
   {
    new_basic_expr_ptr->next = NULL;
    basic_expr_ptr = new_basic_expr_ptr;
   }
 | basic_expr_list COMMA basic_expr
   {
    new_basic_expr_ptr->next = basic_expr_ptr;
    basic_expr_ptr = new_basic_expr_ptr;
   }
 ;

dap_expr:
 relation
   {
    rel_list_ptr->next = NULL;
    dap_expr_ptr->rel_list_ptr = rel_list_ptr;
    dap_expr_ptr->relation_type = Relation;
    rel_list_ptr = NULL;
   }
 | rel_and_list
 | rel_or_list
 :

rel_and_list:
```

```
 relation
  {
   dap_expr_ptr->rel_list_ptr = rel_list_ptr;
   rel_list_ptr = NULL;
  }
 AND relation
  {
   rel_list_ptr->next = NULL;
   dap_expr_ptr->rel_list_ptr->next = rel_list_ptr;
   dap_expr_ptr->relation_type = AndRelation;
   rel_list_ptr = NULL;
  }
 | rel_and_list AND relation
  {
   rel_list_ptr->next = dap_expr_ptr->rel_list_ptr;
   dap_expr_ptr->rel_list_ptr = rel_list_ptr;
   rel_list_ptr = NULL;
  }
 ;

rel_or_list:
 relation
  {
   dap_expr_ptr->rel_list_ptr = rel_list_ptr;
   rel_list_ptr = NULL;
  }
 OR relation
  {
   rel_list_ptr->next = NULL;
   dap_expr_ptr->rel_list_ptr->next = rel_list_ptr;
   dap_expr_ptr->relation_type = OrRelation;
   rel_list_ptr = NULL;
  }
 | rel_and_list OR relation
  {
   rel_list_ptr->next = dap_expr_ptr->rel_list_ptr;
   dap_expr_ptr->rel_list_ptr = rel_list_ptr;
   rel_list_ptr = NULL;
  }
 ;

relation:
  {
   /* This procedure is added to avoid reduce/reduce conflicts in YACC */
   /* memory allocation for rel_list_ptr, simple_expr_ptr          */
   /* are performed here                                           */
   rel_list_ptr = relation_list_alloc();
   simple_expr_ptr = simple_expr1_alloc();
  }
 relation2;
```

```
relation2:
  simple_expr
    {
     rel_list_ptr->simple_expr1_ptr = simple_expr_ptr;
     simple_expr_ptr = NULL;
    }
  | simple_expr rel_op
    {
     rel_list_ptr->simple_expr2_ptr = simple_expr2_alloc();
     rel_list_ptr->simple_expr2_ptr->first_expr = simple_expr_ptr;
     rel_list_ptr->simple_expr2_ptr->rel_operator = rel_operator;
     simple_expr_ptr = simple_expr1_alloc();
    }
  simple_expr
    {
     rel_list_ptr->simple_expr2_ptr->second_expr = simple_expr_ptr ;
     simple_expr_ptr = NULL;
    }
   simple_expr in_op
    {
     rel_list_ptr->simple_expr3_ptr = simple_expr3_alloc();
     rel_list_ptr->simple_expr3_ptr->simple_expr = simple_expr_ptr;
     simple_expr_ptr = NULL;
     rel_list_ptr->simple_expr3_ptr->in_op = in_op;
     dap_range_ptr = dap_range_info_alloc();
     rel_list_ptr->simple_expr3_ptr->dap_range = dap_range_ptr ;
    }
  dap_range
  | simple_expr in_op IDENTIFIER
    {
     if ((search_entity($3) != NULL) || (search_gensub($3) != NULL))
          {
           rel_list_ptr->simple_expr4_ptr = simple_expr4_alloc();
           rel_list_ptr->simple_expr4_ptr->simple_expr = simple_expr_ptr;
           rel_list_ptr->simple_expr4_ptr->in_op = in_op;
           strcpy(rel_list_ptr->simple_expr4_ptr->name_id, $3);
         simple_expr_ptr = NULL;
          }
     else
          proc_eval_error(15,$3);
    }
  ;

simple_expr:
    literal
    {
     strcpy(simple_expr_ptr->lit_array, kms_ptr->dki_evl_ptr.ev_ptr);
     simple_expr_ptr->set_construct_ptr = NULL;
     simple_expr_ptr->indexed_comp_ptr = NULL;
     simple_expr_ptr->funct_appln_ptr = NULL;
    }
```

```
| set_constructor
  {
   simple_expr_ptr->set_construct_ptr = set_construct_ptr;
   simple_expr_ptr->indexed_comp_ptr = NULL;
   simple_expr_ptr->funct_appln_ptr = NULL;
   strcpy(simple_expr_ptr->lit_array, "");
  }
| indexed_component
  {
   simple_expr_ptr->indexed_comp_ptr = indexed_comp_ptr;
   simple_expr_ptr->set_construct_ptr = NULL;
   simple_expr_ptr->funct_appln_ptr = NULL;
   strcpy(simple_expr_ptr->lit_array, "");
  }
| function_application
  {
   simple_expr_ptr->funct_appln_ptr = funct_appln_ptr;
   simple_expr_ptr->indexed_comp_ptr = NULL;
   simple_expr_ptr->set_construct_ptr = NULL;
   strcpy(simple_expr_ptr->lit_array, "");
  }
;

function_application:
  {
   funct_appln_ptr = funct_appln_alloc();
  }
 function_name LP expr_types RP
 ;

function_name:
  COUNT
  {
   funct_appln_ptr->type = COUNT;   /* use token value */
  }
| SUM
  {
   funct_appln_ptr->type = SUM;
  }
| AVG
  {
   funct_appln_ptr->type = AVG;
  }
| MIN
  {
   funct_appln_ptr->type = MIN;
  }
| MAX
  {
   funct_appln_ptr->type = MAX;
  }
```

```
;

expr_types :
 IDENTIFIER
  {
   if ((search_entity($1) != NULL) || (search_gensub($1) != NULL))
        strcpy(funct_appln_ptr->name_id, $1);
   else
        proc_eval_error(15,$1);
  }
 | set_constructor
  {
   funct_appln_ptr->set_construct_ptr = set_construct_ptr;
   funct_appln_ptr->indexed_comp_ptr = NULL;
   strcpy(funct_appln_ptr->name_id, "");
   set_construct_ptr = NULL;
  }
 | indexed_component
  {
   funct_appln_ptr->indexed_comp_ptr = indexed_comp_ptr;
   funct_appln_ptr->set_construct_ptr = NULL;
   strcpy(funct_appln_ptr->name_id, "");
   indexed_comp_ptr = NULL;
  }
 ;

indexed_component:
 dml_name_id LP IDENTIFIER RP
  {
   /* check for the innermost IDENTIFIER */
   indexed_comp_ptr = indexed_component_alloc();
   if ((entptr = search_entity($3)) != NULL)
        indexed_comp_ptr->type = Entity;
   else if ((subptr = search_gensub($3)) != NULL)
        indexed_comp_ptr->type = GenSub;
   else if ((loop_info_ptr != NULL) &&
          (strcmp($3, loop_info_ptr->loop_parameter) == 0))
        {
         indexed_comp_ptr->type = LoopParameter;
         entptr = loop_info_ptr->entptr;
         subptr = loop_info_ptr->subptr;
        }
   else    /* undeclared entity type, subtype or loop parameter */
        {
        proc_eval_error(27, $3);
        break;
        }
   strcpy(indexed_comp_ptr->name_id, $3);
   strcpy(indexed_comp_ptr->parent_name, "");
   indexed_comp_ptr->next = NULL;
```

```
    /* check for each name in kms_ptr->dki_temp_ptr */
    temp_ptr = kms_ptr->dki_temp_ptr;
    while (temp_ptr != NULL)
        {
         /* check if temp_ptr->name is a valid function */
         new_indexed_comp_ptr = indexed_component_alloc();
       new_indexed_comp_ptr->next = indexed_comp_ptr;
       indexed_comp_ptr = new_indexed_comp_ptr;
       strcpy(indexed_comp_ptr->name_id, temp_ptr->name);
         funct_ptr = search_funct(indexed_comp_ptr, entptr, subptr);
         if (funct_ptr == NULL)      /* invalid function name */
           proc_eval_error(26, temp_ptr->name);
         else if (temp_ptr->next != NULL)
          {
           /* check if it is entity type or subtype */
           entptr = funct_ptr->fn_entptr;
           subptr = funct_ptr->fn_subptr;
           if (entptr != NULL)
                indexed_comp_ptr->type = Entity;
           else if (subptr != NULL)
                indexed_comp_ptr->type = GenSub;
           else
                {
                proc_eval_error(15,temp_ptr->name);
                break;
                }
          }
         temp_ptr = temp_ptr->next;
        }
   }
  | dml_name_id LP indexed_component RP
  ;

dml_name_id : IDENTIFIER
   {
    /* stack IDENTIFIERs in kms_ptr->dki_temp_ptr */
    temp_ptr = ident_list_alloc();
    strcpy(temp_ptr->name, $1);
    temp_ptr->next = kms_ptr->dki_temp_ptr;
    kms_ptr->dki_temp_ptr = temp_ptr;
   }
  ;

include_statement:
  INCLUDE
   {
    kms_ptr->dml_statement_ptr = dml_statement_alloc();
    kms_ptr->dml_statement_ptr->type = Include;
    kms_ptr->dml_statement_ptr->dap_expr_ptr = dap_expr_info_alloc();
    kms_ptr->dml_statement_ptr->basic_expr_ptr = NULL;
    dap_expr_ptr = kms_ptr->dml_statement_ptr->dap_expr_ptr;
```

```
  }
 dap_expr INTO indexed_component SEMICOLON
  {
   kms_ptr->dml_statement_ptr->indexed_comp_ptr = indexed_comp_ptr;
  }
 ;

exclude_statement:
 EXCLUDE
  {
   kms_ptr->dml_statement_ptr = dml_statement_alloc();
   kms_ptr->dml_statement_ptr->type = Exclude;
   kms_ptr->dml_statement_ptr->dap_expr_ptr = dap_expr_info_alloc();
   kms_ptr->dml_statement_ptr->basic_expr_ptr = NULL;
   dap_expr_ptr = kms_ptr->dml_statement_ptr->dap_expr_ptr;
  }
 dap_expr FROM indexed_component SEMICOLON
  {
   kms_ptr->dml_statement_ptr->indexed_comp_ptr = indexed_comp_ptr;
  }

 ;

destroy_statement:
 DESTROY
  {
   kms_ptr->dml_statement_ptr = dml_statement_alloc();
   kms_ptr->dml_statement_ptr->type = Destroy;
   kms_ptr->dml_statement_ptr->dap_expr_ptr = dap_expr_info_alloc();
   kms_ptr->dml_statement_ptr->indexed_comp_ptr = NULL;
   kms_ptr->dml_statement_ptr->basic_expr_ptr = NULL;
   dap_expr_ptr = kms_ptr->dml_statement_ptr->dap_expr_ptr;
  }
 dap_expr SEMICOLON
 ;

assignment_statement:
 indexed_component ASSIGN
  {
   kms_ptr->dml_statement_ptr = dml_statement_alloc();
   kms_ptr->dml_statement_ptr->type = Assignment;
   kms_ptr->dml_statement_ptr->basic_expr_ptr = NULL;
   kms_ptr->dml_statement_ptr->indexed_comp_ptr = indexed_comp_ptr;
   kms_ptr->dml_statement_ptr->dap_expr_ptr = dap_expr_info_alloc();
   dap_expr_ptr = kms_ptr->dml_statement_ptr->dap_expr_ptr;
  }
 dap_expr
 ;

move_statement:
 MOVE
```

```
  {
   kms_ptr->dml_statement_ptr = dml_statement_alloc();
   kms_ptr->dml_statement_ptr->type = Move;
   kms_ptr->dml_statement_ptr->dap_expr_ptr = dap_expr_info_alloc();
   kms_ptr->dml_statement_ptr->indexed_comp_ptr = NULL;
   kms_ptr->dml_statement_ptr->basic_expr_ptr = NULL;
   dap_expr_ptr = kms_ptr->dml_statement_ptr->dap_expr_ptr;
  }
 dap_expr move_statement2 SEMICOLON
 ;

move_statement2 :
  move_from
 | move_to
 | move_from move_to
 ;

move_from:
 FROM name1_list
 :

move_to:
 INTO create_part
 :

procedure_call:
  {
   kms_ptr->dml_statement_ptr = dml_statement_alloc();
   kms_ptr->dml_statement_ptr->basic_expr_ptr = basic_expr_ptr;
   kms_ptr->dml_statement_ptr->dap_expr_ptr = NULL;
   kms_ptr->dml_statement_ptr->indexed_comp_ptr = NULL;
  }
 procedure_name LP basic_expr_list RP SEMICOLON
 ;

procedure_name:
 PRINT
  {
   kms_ptr->dml_statement_ptr->type = Print;
  }
 | PRINT_LINE
  {
   kms_ptr->dml_statement_ptr->type = PrintLine;
  }
 ;

loop_statement:
  real_loop SEMICOLON
 | IDENTIFIER COLON real_loop IDENTIFIER SEMICOLON
 | IDENTIFIER COLON real_loop SEMICOLON
 ;
```

```
real_loop:
    {
     loop_info_ptr = loop_info_alloc();
     kms_ptr->loop_info_ptr = loop_info_ptr;
    }
  iteration_clause basic_loop end_loop
  ;

iteration_clause:
  iteration_body
    {
      loop_info_ptr->order_comp_ptr = NULL;
    }
  | iteration_body BY order_component_list
  ;

iteration_body:
  for_clause IDENTIFIER
    {
      strcpy(loop_info_ptr->loop_parameter, $2);
      domain_ptr = &(loop_info_ptr->domain);
    }
  IN domain
  ;

for_clause:
    FOR
  | FOR EACH
  ;

domain:
  loop_expr
    {
     domain_ptr->dap_expr_ptr = NULL;
    }
  | loop_expr WHERE
    {
     dap_expr_ptr = dap_expr_info_alloc();
     domain_ptr->dap_expr_ptr = dap_expr_ptr;
    }
  dap_expr
  ;

loop_expr:
    indexed_component
    {
     domain_ptr->indexed_comp_ptr = indexed_comp_ptr;
     strcpy(domain_ptr->name,"");
    }
  | IDENTIFIER
    {
```

```
    /* IDENTIFIER must be entity type or subtype */
    loop_info_ptr->entptr = search_entity($1);
    loop_info_ptr->subptr = search_gensub($1);
    if ((loop_info_ptr->entptr != NULL) || (loop_info_ptr->subptr != NULL))
         {
          domain_ptr->indexed_comp_ptr = NULL;
          strcpy(domain_ptr->name,$1);
         }
    else
         proc_eval_error(15,$1);
   }
   :

order_component_list:
  order_component
   {
    order_comp_ptr->next = NULL;
    loop_info_ptr->order_comp_ptr = order_comp_ptr;
    order_comp_ptr = NULL;
   }
  | order_component_list COMMA order_component
   {
    order_comp_ptr->next = loop_info_ptr->order_comp_ptr;
    loop_info_ptr->order_comp_ptr = order_comp_ptr;
    order_comp_ptr = NULL;
   }
  ;

order_component:
   {
    order_comp_ptr = order_comp_list_alloc();
   }
  sort_order indexed_component
   {
    order_comp_ptr->indexed_comp_ptr = indexed_comp_ptr;
    indexed_comp_ptr = NULL;
   }
  ;

sort_order:
  ASCENDING
   {
    order_comp_ptr->sort_order = ASCENDING;
   }
  | DESCENDING
   {
    order_comp_ptr->sort_order = DESCENDING;
   }
  | /* empty */
   {
    order_comp_ptr->sort_order = ASCENDING;
```

```
    }
  ;

basic_loop:
  sequence_of_statements
  | LOOP sequence_of_statements
  :

sequence_of_statements:
  dml_statement2
    {
    dml_statement2_list_ptr = dml_statement2_list_alloc();
    dml_statement2_list_ptr->dml_statement2_ptr = kms_ptr->dml_statement_ptr;
    dml_statement2_list_ptr->next = NULL;
    loop_info_ptr->dml_statement2_list_ptr = dml_statement2_list_ptr;
    kms_ptr->dml_statement_ptr = NULL;
    }
  | sequence_of_statements dml_statement2
    {
    dml_statement2_list_ptr = dml_statement2_list_alloc();
    dml_statement2_list_ptr->dml_statement2_ptr = kms_ptr->dml_statement_ptr;
    dml_statement2_list_ptr->next = loop_info_ptr->dml_statement2_list_ptr;
    loop_info_ptr->dml_statement2_list_ptr = dml_statement2_list_ptr;
    kms_ptr->dml_statement_ptr = NULL;
    }
  ;

end_loop:
  END
  | END LOOP
  ;

/*
name1_list:  name1
        | name1_list COMMA name1
  ;

name1:    name_id
        | name_id  DOT name_id
  ;
*/

rel_op:
  EQ
    {
    rel_operator = EQ;          /* use token value */
    }
  | NE
    {
    rel_operator = NE;
    }
```

```
  | LT
    {
     rel_operator = LT;
    }
  | LE
    {
     rel_operator = LE;
    }
  | GT
    {
     rel_operator = GT;
    }
  | GE
    {
     rel_operator = GE;
    }
  ;

in_op:
  IN
    {
     in_op = INOp;
    }
  | NOT IN
    {
     in_op = NINOp;
    }
  ;

dap_range:
  INTEGER_LITERAL ELIPSES INTEGER_LITERAL
    {
     dap_range_ptr->range_type = Integer;
     strcpy(dap_range_ptr->first_value, $1);
     strcpy(dap_range_ptr->second_value, $3);
    }
  | FLOAT_LITERAL ELIPSES FLOAT_LITERAL
    {
     dap_range_ptr->range_type = Float;
     strcpy(dap_range_ptr->first_value, $1);
     strcpy(dap_range_ptr->second_value, $3);
    }
  ;

%%



f_kernel_mapping_system ()
  {
    reset_variables();
```

```
   /* alloc and init dap_kms_info struct */
   kms_ptr = dap_kms_info_alloc();
   kms_ptr->dki_temp_ptr = NULL;
   kms_ptr->dki_id_ptr = NULL;
   kms_ptr->dki_overfirst_ptr = NULL;
   kms_ptr->dki_ev_ptr = NULL;
   kms_ptr->dki_create = NULL;
   kms_ptr->dki_req_ptr = NULL;
   kms_ptr->dki_cel_ptr = NULL;
   kms_ptr->dki_create_ovrptr = NULL;
  if (cuser_dap_ptr->ui_li_type.li_dap.dap_operation != CreateDB)
   {
   cuser_dap_ptr->ui_li_type.li_dap.dpi_kms_data.ki_d_kms = kms_ptr;
   dml_req_len = cuser_dap_ptr->ui_li_type.li_dap.dpi_dml_tran.
                                ti_curr_req.ri_dap_req->dri_req_len;
/*
   dml_info_ptr = &(cuser_dap_ptr->ui_li_type.li_dap);
   strcpy(db.dml_info_ptr->dpi_curr_db.cdi_dbname);
*/
   }

  mem_ptr = cuser_dap_ptr->ui_li_type.li_dap.dpi_dml_tran.
                                ti_curr_req.ri_dap_req->dri_req;
#ifdef  DYacFlag
        printf ("calling yyparse \n");
#endif

  yyparse();

#ifdef  DYacFlag
        printf ("returning from yyparse \n");
#endif

  /* reset all boolean and counter variables */
  kms_info_cleanup();
#ifdef  DYacFlag
        printf ("Exit parser \n");
#endif
  }

yyerror (s)
  char  *s;
  {
  if (cuser_dap_ptr->ui_li_type.li_dap.dap_operation == CreateDB)
   {
   cuser_dap_ptr->ui_li_type.li_dap.dap_error = ErrCreateDB;
   printf ("Correct error before re-submitting that DB Description file \n");
   /* free all the malloc'd variables in the current schema */
   schema_cleanup();
   }
  else
```

```
   {
    cuser_dap_ptr->ui_li_type.li_dap.dap_error = ErrReqTran;
    kms_info_cleanup();
   }

  /* reset all boolean and counter variables */
  reset_variables();
  printf ("\n%s \n", s);
  }


schema_cleanup()
{
}

kms_info_cleanup()
{
}

reset_variables()
{
   last_funct_ptr = NULL;
}
```

2. FILE kms.c

```
#include <stdio.h>
#include <strings.h>
#include "flags.def"
#include "licommdata.def"
#include "struct.def"
#include "dap.ext"
#include "kms.ext"




proc_eval_error(num,str_in)
int   num;
char *str_in;

{
 switch (num)
  {
   case  1: printf("Error - duplicate constant name:");
            break;
   case  2: printf("Error - duplicate non_entity type name:");
            break;
   case  3: printf("Error - duplicate entity type name:");
```

```
            break;
case  4: printf("Error - cannot create new type. Base type undefined:");
            break;
case  5: printf("Error - attribute name must be unique:");
            break;
case  6: printf("Error - undeclared function identifier:");
            break;
case  7: printf("Error - nonentity subtype name must be unique:");
            break;
case  8: printf("Error - incompatible types must be nonentity type:");
            break;
case  9: printf("Error - entity type must have been declared previously:");
            break;
case 10: printf("Error - undeclared subtype name:");
            break;
case 11: printf("Error - incompatible types must be entity type:");
            break;
case 12: printf("Error - duplicate entity subtype name:");
            break;
case 13: printf("Error - duplicate identifier in list:");
            break;
case 14: printf("Error - overlap type must be terminal subtype:");
            break;
case 15: printf("Error - undefined entity type or subtype:");
            break;
case 16: printf("Error - incompatible derived type declaration:");
            break;
case 17: printf("Error - incompatible subtype declaration:");
            break;
case 18: printf("Error - incompatible assignment:");
            break;
case 19: printf("Error - identifier must be previously declared:");
            break;
case 20: printf("Error - overlap types must have same base type:");
            break;
case 21: printf("Error - non-terminal type entity in create:");
            break;
case 22: printf("Error - more than one entity type name in create:");
            break;
case 23: printf("Error - no default value for enumeration type attribute:");
            break;
case 24: printf("Error - value must be declared for entity type attribute:");
            break;
case 25: printf("Error - overlap constraint not declared:");
            break;
case 26: printf("Error - invalid function name:");
            break;
case 27: printf("Error - undefined entity type, subtype or loop parameter:");
            break;
default: printf("Error - error code: %d:", num);
            break;
```

```
  }   /* end switch */
  printf(" %s\n", str_in);
}

put_dki_ent_non(enn_type, str_val)

char  enn_type;
char  *str_val;

{
   struct ent_value      *ent_value_alloc();
   char *var_str_alloc();

        kms_ptr->dki_ent_non.enn_type = enn_type;
      switch (enn_type)
         {
         case 'f' : kms_ptr->dki_ent_non.enn_total_length = FLTLength;
                    break;
         case 'b' :
         case 'i' : kms_ptr->dki_ent_non.enn_total_length = INTLength;
                    break;
         default  : ;
         }
        kms_ptr->dki_ent_non.enn_range = FALSE;
        kms_ptr->dki_ent_non.enn_num_values = 1;
        kms_ptr->dki_ent_non.enn_constant = TRUE;
        kms_ptr->dki_ent_non.enn_value = ent_value_alloc();
        kms_ptr->dki_ent_non.enn_value->ev_value =
                  var_str_alloc( strlen(str_val) + 1 );
        strcpy(kms_ptr->dki_ent_non.enn_value->ev_value, str_val);
        kms_ptr->dki_ent_non.enn_value->next = NULL;
}


put_dki_funct(fn_type)

char  fn_type;

{
     kms_ptr->dki_funct.fn_type = fn_type;
     switch (fn_type)
         {
         case 'f' : kms_ptr->dki_funct.fn_total_length = FLTLength;
                    break;
         case 'b' :
         case 'i' : kms_ptr->dki_funct.fn_total_length = INTLength;
                    break;
         default  : kms_ptr->dki_funct.fn_total_length = 0;
                    break;
         }
     kms_ptr->dki_funct.fn_range = FALSE;
```

```
    kms_ptr->dki_funct.fn_set = FALSE;
    kms_ptr->dki_funct.fn_num_value = 0;
    kms_ptr->dki_funct.fn_value = NULL;
    kms_ptr->dki_funct.fn_entptr = NULL;
    kms_ptr->dki_funct.fn_subptr = NULL;
    kms_ptr->dki_funct.fn_nonentptr = NULL;
    kms_ptr->dki_funct.fn_nonsubptr = NULL;
    kms_ptr->dki_funct.fn_nonderptr = NULL;
    kms_ptr->dki_funct.next = NULL;
    kms_ptr->dki_funct.fn_entnull = FALSE;
    kms_ptr->dki_funct.fn_unique = FALSE;
}

add_ent_non_node(enn_name)

char  enn_name[ENLength + 1];

{
   struct fun_dbid_node *db_ptr;
   struct ent_non_node  *ent_non_node_alloc(),
                        *enn_ptr;

        db_ptr = cuser_dap_ptr->ui_li_type.li_dap.dpi_curr_db.cdi_db.dn_fun;
        enn_ptr = ent_non_node_alloc();
        strcpy(enn_ptr->enn_name, enn_name);
        enn_ptr->enn_type = kms_ptr->dki_ent_non.enn_type;
        enn_ptr->enn_total_length = kms_ptr->dki_ent_non.enn_total_length;
        enn_ptr->enn_range = kms_ptr->dki_ent_non.enn_range;
        enn_ptr->enn_num_values = kms_ptr->dki_ent_non.enn_num_values;
        enn_ptr->enn_value = kms_ptr->dki_ent_non.enn_value;
        kms_ptr->dki_ent_non.enn_value = NULL;
        enn_ptr->enn_constant = kms_ptr->dki_ent_non.enn_constant;
        enn_ptr->enn_next_node = db_ptr->fdn_nonentptr;
        db_ptr->fdn_nonentptr = enn_ptr;
        db_ptr->fdn_num_nonent++;
}


struct gen_sub_node *new_gen_sub_node(temp_value)

char temp_value[ENLength + 1];

{
  struct fun_dbid_node  *db_ptr;
  struct gen_sub_node   *gen_sub_node_alloc(),
                        *gen_ptr;

        db_ptr = cuser_dap_ptr->ui_li_type.li_dap.dpi_curr_db.cdi_db.dn_fun;
        gen_ptr = gen_sub_node_alloc();
        strcpy(gen_ptr->gsn_name, temp_value);
        gen_ptr->gsn_num_funct = 0;
```

```
        gen_ptr->gsn_terminal = TRUE;
        gen_ptr->gsn_entptr = NULL;
        gen_ptr->gsn_num_ent = 0;
        gen_ptr->gsn_ftnptr = NULL;
        gen_ptr->gsn_subptr = NULL;
        gen_ptr->gsn_num_sub = 0;
        gen_ptr->gsn_next_genptr = db_ptr->fdn_subptr;
        db_ptr->fdn_subptr = gen_ptr;
      db_ptr->fdn_num_gen++;
        return(gen_ptr);
}


compare_non_node_type(temp_value, serror)

char temp_value[ENLength + 1];
int  serror;

{
  struct fun_dbid_node *db_ptr;
  struct ent_non_node  *enn_ptr;
  struct sub_non_node  *snn_ptr;
  struct der_non_node  *dnn_ptr;
  int                  in;

        db_ptr = cuser_dap_ptr->ui_li_type.li_dap.dpi_curr_db.cdi_db.dn_fun;
        enn_ptr = db_ptr->fdn_nonentptr;
        snn_ptr = db_ptr->fdn_nonsubptr;
        dnn_ptr = db_ptr->fdn_nonderptr;

        in = FALSE;
        while ((enn_ptr != NULL) && (!in))
         {
          if (strcmp(enn_ptr->enn_name, temp_value) == FALSE)
            {
               in = TRUE;
               if (enn_ptr->enn_type != kms_ptr->dki_ent_non.enn_type)
                 proc_eval_error(serror,temp_value);
            }
          else
               enn_ptr = enn_ptr->enn_next_node;
         }

        while ((snn_ptr != NULL) && (!in))
         {
          if (strcmp(snn_ptr->snn_name, temp_value) == FALSE)
            {
               in = TRUE;
               if (snn_ptr->snn_type != kms_ptr->dki_ent_non.enn_type)
                 proc_eval_error(serror,temp_value);
            }
```

```
            else
                snn_ptr = snn_ptr->snn_next_node;
          }

        while ((dnn_ptr != NULL) && (!in))
          {
           if (strcmp(dnn_ptr->dnn_name, temp_value) == FALSE)
             {
                in = TRUE;
                if (dnn_ptr->dnn_type != kms_ptr->dki_ent_non.enn_type)
                  proc_eval_error(serror.temp_value);
             }
           else
                dnn_ptr = dnn_ptr->dnn_next_node;
          }

        if (!in)
          proc_eval_error(serror.temp_value);
}

add_range_non_node(enn_type, str1, str2)

char  enn_type;
char  *str1, *str2;

{
   struct ent_value      *ent_value_alloc(),
                         *entval_ptr;
   char *var_str_alloc();

    /* update kms_info */
        kms_ptr->dki_ent_non.enn_type = enn_type;
      switch (enn_type)
         {
          case 'f' : kms_ptr->dki_ent_non.enn_total_length = FLTLength;
                     break;
          case 'i' : kms_ptr->dki_ent_non.enn_total_length = INTLength;
                     break;
          }
        kms_ptr->dki_ent_non.enn_range = TRUE;
        kms_ptr->dki_ent_non.enn_num_values = 2;
        kms_ptr->dki_ent_non.enn_constant = FALSE;

    /* create value node for first integer */
        kms_ptr->dki_ent_non.enn_value = ent_value_alloc();
        kms_ptr->dki_ent_non.enn_value->ev_value =
                                  var_str_alloc( strlen(str1) + 1);
        strcpy(kms_ptr->dki_ent_non.enn_value->ev_value, str1);

    /* create value node for second integer */
        entval_ptr = ent_value_alloc();
```

```
        entval_ptr->ev_value = var_str_alloc( strlen(str2) + 1);
        strcpy(entval_ptr->ev_value, str2);
        entval_ptr->next = NULL;
        kms_ptr->dki_ent_non.enn_value->next = entval_ptr;
}


free_id_list()

{
  struct ident_list  *ident_ptr,
                     *next_ident_ptr;

  ident_ptr = kms_ptr->dki_id_ptr;
  while (ident_ptr != NULL )
    {
        next_ident_ptr = ident_ptr->next;
        free(ident_ptr);
        ident_ptr = next_ident_ptr;
    }
  kms_ptr->dki_id_ptr = NULL;
}

free_temp_list()

{
  struct ident_list  *ident_ptr,
                     *next_ident_ptr;

  ident_ptr = kms_ptr->dki_temp_ptr;
  while (ident_ptr != NULL )
    {
        next_ident_ptr = ident_ptr->next;
        free(ident_ptr);
        ident_ptr = next_ident_ptr;
    }
  kms_ptr->dki_temp_ptr = NULL;
}

free_name1_list()

{
  struct ident_list  *ident_ptr,
                     *next_ident_ptr;

  ident_ptr = kms_ptr->dki_name1_ptr;
  while (ident_ptr != NULL )
    {
        next_ident_ptr = ident_ptr->next;
        free(ident_ptr);
        ident_ptr = next_ident_ptr;
```

```
    }
  kms_ptr->dki_name1_ptr = NULL;
}


find_base_type(genptr, base_name)

struct gen_sub_node     *genptr;
char   *base_name;

 {
  if (genptr->gsn_subptr != NULL)
    find_base_type(genptr->gsn_subptr->subptr, base_name);
  else if (genptr->gsn_entptr != NULL)
    strcpy(base_name, genptr->gsn_entptr->entptr->en_name);
  else
    strcpy(base_name, genptr->gsn_name);
 }
```

3. FILE : dml.c

```
#include <stdio.h>
#include <strings.h>
#include <ctype.h>
#include "flags.def"
#include "licommdata.def"
#include "struct.def"
#include "dap.ext"
#include "kms.ext"



proc_create_ent_type(entptr)

struct ent_node     *entptr;

 {
   struct dap_create_list   *dap_create_list_alloc();
   struct dap_av_pair_list  *build_funct_av_pair(),
                            *build_key_av_pair(),
                            *av_ptr;
   struct ident_list        *temp_ptr;
   int    in;

#ifdef EnExFlag
   printf("Enter proc_create_ent_type\n");
#endif
   /* Check if first name_id is entity type */
```

```
    temp_ptr = kms_ptr->dki_temp_ptr;
    in = FALSE;
    while ((entptr != NULL) && (!in))
     {
      if (strcmp(entptr->en_name, temp_ptr->name) == FALSE)
            in = TRUE;
         else
            entptr = entptr->en_next_ent;
      }

    if (in)
      {
         if (entptr->en_terminal == FALSE)
           /* must be terminal type for create */
           proc_eval_error(21,temp_ptr->name);

         else if (temp_ptr->next != NULL)
           /* not more than one name in list when creating entity type */
           proc_eval_error(22.temp_ptr->name);

         else    /* build create_ent_list */
           {
           free_dki_create();
           kms_ptr->dki_create = dap_create_list_alloc();
           kms_ptr->dki_create->req_type = Insert;
           strcpy(kms_ptr->dki_create->en_name, entptr->en_name);
           av_ptr = build_key_av_pair(entptr);
           av_ptr->next = build_funct_av_pair(entptr->en_ftnptr);
           kms_ptr->dki_create->av_pair_ptr = av_ptr;
           kms_ptr->dki_create->next =  NULL;
           }
         }
#ifdef EnExFlag
   printf("Exit proc_create_ent_type\n");
#endif
    return(in);
  }  /* end proc_create_ent_type */

proc_create_sub_type(genptr)

struct gen_sub_node  *genptr;

 {
    int    in;
    struct fun_dbid_node    *db_ptr;
    struct overlap_node     *ovrptr;
    struct gen_sub_node     *subptr;
    struct sub_node_list    *snlptr;
    struct create_ent_list  *cel_ptr,
                             *create_ent_list_alloc();
    struct sub_node_list    *sub_node_list_alloc();
```

```
    struct ident_list       *temp_ptr;

#ifdef EnExFlag
   printf("Enter proc_create_sub_type\n");
#endif

    free_dki_create();
    kms_ptr->dki_cel_ptr = NULL;

    /* ensure multiple name ids are overlap sub node type */
    /* skip if there is only one name id in the list      */
    temp_ptr = kms_ptr->dki_temp_ptr;
    if (temp_ptr->next != NULL)
        {
        db_ptr = cuser_dap_ptr->ui_li_type.li_dap.dpi_curr_db.cdi_db.dn_fun;
        ovrptr = db_ptr->fdn_ovrptr;
        in = FALSE;
        while (ovrptr != NULL && (!in))
          {
          snlptr = ovrptr->snlptr;
          while (snlptr != NULL && (!in))
            {
            if (strcmp(snlptr->subptr->gsn_name, temp_ptr->name) == 0)
                {
                in = TRUE;
                kms_ptr->dki_create_ovrptr = ovrptr;
                }
            else
              snlptr = snlptr->next;
            }
          ovrptr = ovrptr->next;
          }
        if (!in)   /* overlap constraint not declared */
          proc_eval_error(25,temp_ptr->name);
        else
          {
          /* check the all remaining name_ids */
          temp_ptr = temp_ptr -> next;
          while (temp_ptr != NULL)
            {
            snlptr = kms_ptr->dki_create_ovrptr->snlptr;
            in = FALSE;
            while (snlptr != NULL && (!in))
                {
                if (strcmp(temp_ptr->name, snlptr->subptr->gsn_name) == 0)
                  in = TRUE;
                else
                  snlptr = snlptr->next;
                }
            if (!in)   /* overlap constraint not declared */
                proc_eval_error(25,temp_ptr->name);
```

```
            temp_ptr = temp_ptr->next;
           }
         }
       }
   temp_ptr = kms_ptr->dki_temp_ptr;
   while (temp_ptr != NULL)
    {
       subptr = genptr;   /* rewind subptr */
       in = FALSE;
       while ((subptr != NULL) && (!in))
           {
            if (strcmp(subptr->gsn_name, temp_ptr->name) == FALSE)
            in = TRUE;
            else
                 subptr = subptr->gsn_next_genptr;
           }

       if (!in)
          /* undeclared entity type or subtype */
          proc_eval_error(15,temp_ptr->name);

       else if (subptr->gsn_terminal == FALSE)
          /* must be terminal type for create */
          proc_eval_error(21,temp_ptr->name);

       else
         {
          cel_ptr = create_ent_list_alloc();
          cel_ptr->enl_ptr = NULL;
          cel_ptr->snl_ptr = sub_node_list_alloc();
          cel_ptr->snl_ptr->subptr = subptr;
          cel_ptr->snl_ptr->next = NULL;
          cel_ptr->next = kms_ptr->dki_cel_ptr;
          kms_ptr->dki_cel_ptr = cel_ptr;
         }

       temp_ptr = temp_ptr->next;
    } /* end while temp_ptr */

    build_create_list();

#ifdef EnExFlag
  printf("Exit proc_create_sub_type\n");
#endif

 } /* end proc_create_sub_type */

build_create_list()

 {
   struct sub_node_list    *snl_ptr;
```

```
    struct ent_node_list    *enl_ptr;
    struct ent_node          *overlap_entptr;
    struct gen_sub_node     *overlap_subptr;
    struct create_ent_list  *cel_ptr,
                             *new_cel_ptr,
                             *temp_cel_ptr,
                             *create_ent_list_alloc();

#ifdef EnExFlag
   printf("Enter build_create_list\n");
#endif


    new_cel_ptr = NULL;
    overlap_entptr = NULL;
    overlap_subptr = NULL;
    cel_ptr = kms_ptr->dki_cel_ptr;
    while (cel_ptr != NULL)
     {
        snl_ptr = cel_ptr->snl_ptr;
        enl_ptr = cel_ptr->enl_ptr;
        while (snl_ptr != NULL)
         {
          /* skip if it is the overlapping base type   */
          /* it will be added last to avoid duplicates */
          if (strcmp(snl_ptr->subptr->gsn_name,
                kms_ptr->dki_create_ovrptr->base_type_name) == 0)
              overlap_subptr = snl_ptr->subptr;
          else
            {
              build_sub_create_list(snl_ptr->subptr);
              if ((snl_ptr->subptr->gsn_entptr != NULL) ||
                 (snl_ptr->subptr->gsn_subptr != NULL))
                 {
                  temp_cel_ptr = create_ent_list_alloc();
                  temp_cel_ptr->enl_ptr = snl_ptr->subptr->gsn_entptr;
                  temp_cel_ptr->snl_ptr = snl_ptr->subptr->gsn_subptr;
                  temp_cel_ptr->next = new_cel_ptr;
                  new_cel_ptr = temp_cel_ptr;
                 }
            }
          snl_ptr = snl_ptr->next;
         }  /* end while snl_ptr */

        while (enl_ptr != NULL)
         {
          /* skip if it is the overlapping base type   */
          /* it will be added last to avoid duplicates */
          if (strcmp(enl_ptr->entptr->en_name,
                kms_ptr->dki_create_ovrptr->base_type_name) == 0)
              overlap_entptr = enl_ptr->entptr;
```

```
            else
                  build_ent_create_list(enl_ptr->entptr);
            enl_ptr = enl_ptr->next;
           } /* end while enl_ptr */

          cel_ptr = cel_ptr->next;
      } /* end while cel_ptr */

   free_dki_cel_ptr();
   /* recursively call build_create_list() when there is more supertypes */
   if (new_cel_ptr != NULL)
     {
         kms_ptr->dki_cel_ptr = new_cel_ptr;
         build_create_list();
     }
   else
     {
        /* Now, build the overlap base entity */
        if (overlap_entptr != NULL)
          build_ent_create_list(overlap_entptr);
        else if (overlap_subptr != NULL)
          build_sub_create_list(overlap_subptr);
     }


#ifdef EnExFlag
   printf("Exit build_create_list\n");
#endif

 }  /* end build_create_list */

build_ent_create_list(entptr)

struct ent_node *entptr;

 {
   struct dap_create_list   *dap_create_list_alloc(),
                            *dcl_ptr;
   struct dap_av_pair_list  *build_funct_av_pair(),
                            *build_key_av_pair(),
                            *av_ptr;

#ifdef EnExFlag
   printf("Enter build_ent_create_list\n");
#endif
        dcl_ptr = dap_create_list_alloc();
        dcl_ptr->req_type = Insert;
        strcpy(dcl_ptr->en_name, entptr->en_name);
        av_ptr = build_key_av_pair(entptr);
        av_ptr->next = build_funct_av_pair(entptr->en_ftnptr);
        dcl_ptr->av_pair_ptr = av_ptr;
```

```
        dcl_ptr->next = kms_ptr->dki_create;
        kms_ptr->dki_create = dcl_ptr;

#ifdef EnExFlag
   printf("Exit build_ent_create_list\n");
#endif
 }

build_sub_create_list(subptr)

struct gen_sub_node     *subptr;

 {
   struct dap_create_list   *dap_create_list_alloc(),
                            *dcl_ptr;
   struct dap_av_pair_list  *build_funct_av_pair(),
                            *build_sub_key_av_pair(),
                            *last_av_ptr,
                            *av_ptr;

#ifdef EnExFlag
   printf("Enter build_sub_create_list\n");
#endif

        dcl_ptr = dap_create_list_alloc();
        dcl_ptr->req_type = Insert;
        strcpy(dcl_ptr->en_name, subptr->gsn_name);
        av_ptr = build_sub_key_av_pair(subptr);
        last_av_ptr = av_ptr;
        while (last_av_ptr->next != NULL)
          last_av_ptr = last_av_ptr->next;
        last_av_ptr->next =
            build_funct_av_pair(subptr->gsn_ftnptr);
        dcl_ptr->av_pair_ptr = av_ptr;
        dcl_ptr->next = kms_ptr->dki_create;
        kms_ptr->dki_create = dcl_ptr;

#ifdef EnExFlag
   printf("Exit build_sub_create_list\n");
#endif
 }

struct dap_av_pair_list  *build_key_av_pair(entptr)

struct ent_node     *entptr;

 {
   struct dap_av_pair_list  *dap_av_pair_list_alloc(),
                            *av_ptr;
   struct ent_value        *ent_value_alloc();
   char   *var_str_alloc();
```

```
    av_ptr = dap_av_pair_list_alloc();
    strcpy(av_ptr->name, entptr->en_name);
    av_ptr->ftnptr = NULL;
    av_ptr->num_value = 1;
    av_ptr->valptr = ent_value_alloc();
    av_ptr->valptr->ev_value = var_str_alloc(INTLength);
    num_to_str(entptr->en_last_ent_id + 1, av_ptr->valptr->ev_value);
    av_ptr->valptr->next = NULL;
    av_ptr->next = NULL;

    return(av_ptr);
  }

struct dap_av_pair_list  *build_sub_key_av_pair(subptr)

struct gen_sub_node  *subptr;

  {
    struct dap_av_pair_list  *dap_av_pair_list_alloc(),
                             *new_av_ptr,
                             *last_av_ptr,
                             *av_ptr;
    struct ent_value        *ent_value_alloc();
    struct ent_node_list     *enl_ptr;
    struct sub_node_list     *snl_ptr;

#ifdef EnExFlag
    printf("Enter build_sub_key_av_pair\n");
#endif

    av_ptr = NULL;
    enl_ptr = subptr->gsn_entptr;
    snl_ptr = subptr->gsn_subptr;

    while (enl_ptr != NULL)
      {
        new_av_ptr = build_key_av_pair(enl_ptr->entptr);
        last_av_ptr = new_av_ptr;
        while (last_av_ptr->next != NULL)
          last_av_ptr = last_av_ptr->next;
        last_av_ptr->next = av_ptr;
        av_ptr = new_av_ptr;
        enl_ptr = enl_ptr->next;
      }

    while (snl_ptr != NULL)
      {
        new_av_ptr = build_sub_key_av_pair(snl_ptr->subptr);
        last_av_ptr = new_av_ptr;
        while (last_av_ptr->next != NULL)
          last_av_ptr = last_av_ptr->next;
```

```
        last_av_ptr->next = av_ptr;
        av_ptr = new_av_ptr;
        snl_ptr = sr_ _ptr->next;
      }

#ifdef EnExFlag
   printf("Exit build_sub_key_av_pair\n");
#endif

   return(av_ptr);
  }

struct dap_av_pair_list *build_funct_av_pair(ftnptr)

struct function_node   *ftnptr;

  {
   struct dap_av_pair_list *dap_av_pair_list_alloc(),
                           *av_ptr,
                           *last_av_ptr;

   last_av_ptr = NULL;
   while (ftnptr !=  NULL)
    {
        av_ptr = dap_av_pair_list_alloc();
        strcpy(av_ptr->name, ftnptr->fn_name);
        av_ptr->ftnptr = ftnptr;
        av_ptr->num_value = 0;
        av_ptr->valptr = NULL;
        av_ptr->next = last_av_ptr;
        last_av_ptr = av_ptr;
        ftnptr = ftnptr->next;
    }  /*end while ftnptr */

   return(av_ptr);
  }  /* end build_funct_av_pair */

build_req_line_list()
  {
   struct dap_create_list  *dcl_ptr;
   struct req_line_list    *req_ptr,
                           *r_ptr,
                           *last_req_ptr,
                           *append_attr_value(),
                           *req_line_list_alloc();
   struct dap_av_pair_list *av_ptr;
   int    en_len;         /* length of en_name */
   int    i;              /* index */

   free_dki_req_ptr();
   dcl_ptr = kms_ptr->dki_create;
```

```
while (dcl_ptr != NULL)
 {
     if (kms_ptr->dki_req_ptr == NULL)
      {
       req_ptr = req_line_list_alloc();
       kms_ptr->dki_req_ptr = req_ptr;
      }
     else
      {
       last_req_ptr->next = req_line_list_alloc():
       req_ptr = last_req_ptr->next;
      }
     last_req_ptr = req_ptr;
     if (dcl_ptr->req_type == Insert)
       strcpy(req_ptr->req_line, "INSERT(<FILE,");
     else
       strcpy(req_ptr->req_line, "RETRIEVE(<FILE,");
     en_len = strlen(dcl_ptr->en_name);
     for (i = 1: i < en_len; i++)
        dcl_ptr->en_name[i] = tolower(dcl_ptr->en_name[i]);
     strcat(req_ptr->req_line.dcl_ptr->en_name):
     av_ptr = dcl_ptr->av_pair_ptr;
     while (av_ptr != NULL)
      {
       r_ptr = req_ptr;
       while (r_ptr != NULL)
        {
         strcat(r_ptr->req_line,">,<");
         strcat(r_ptr->req_line,av_ptr->name);
         strcat(r_ptr->req_line,",");
         r_ptr = r_ptr->next;
        }
       if (av_ptr->num_value > 0)
         last_req_ptr =  append_attr_value(req_ptr.av_ptr->valptr);

       else if (av_ptr->ftnptr->fn_num_value > 0)
          /* use default value */
         last_req_ptr =
                append_attr_value(req_ptr,av_ptr->ftnptr->fn_value);
       else
        {
         r_ptr = req_ptr;
         while (r_ptr != NULL)
          {
           /* provide system default value */
           switch (get_base_fun_type(av_ptr->ftnptr))
                {
                 case 'i': strcat(r_ptr->req_line,"0");
                           break;
                 case 'f': strcat(r_ptr->req_line,"0.0");
                           break;
```

```
                    case 's': strcat(r_ptr->req_line,"*****");
                              break;
                    case 'e': proc_eval_error(23,av_ptr->name):
                              break;
                    case 'E': if (av_ptr->ftnptr->fn_entnull)
                                   strcat(r_ptr->req_line,"0");
                              else
                                   proc_eval_error(24,av_ptr->name);
                              break;
                   }
                  r_ptr = r_ptr->next:
                 }
              }
            av_ptr = av_ptr->next;
           }
         r_ptr = req_ptr;
         while (r_ptr != NULL)
           {
            strcat(r_ptr->req_line,">)");
            r_ptr = r_ptr->next;
           }
         dcl_ptr = dcl_ptr->next;
     }
  }


free_dki_create()
  {
   struct dap_create_list*dcl_ptr,
                          *next_dcl_ptr;

   dcl_ptr = kms_ptr->dki_create;
   while (dcl_ptr != NULL)
     {
      next_dcl_ptr = dcl_ptr->next;
      free(dcl_ptr);
      dcl_ptr = next_dcl_ptr;
     }
   kms_ptr->dki_create = NULL;
  }



free_dki_req_ptr()
  {
   struct req_line_list   *req_ptr,
                          *next_req_ptr;

   req_ptr = kms_ptr->dki_req_ptr;
   while (req_ptr != NULL)
     {
```

```
      next_req_ptr = req_ptr->next;
      free(req_ptr);
      req_ptr = next_req_ptr;
     }
    kms_ptr->dki_req_ptr = NULL;
  }



free_dki_cel_ptr()
  {
    struct create_ent_list *cel_ptr,
                           *next_cel_ptr;

    cel_ptr = kms_ptr->dki_cel_ptr;
    while (cel_ptr != NULL)
      {
       next_cel_ptr = cel_ptr->next;
       free(cel_ptr);
       cel_ptr = next_cel_ptr;
      }
    kms_ptr->dki_cel_ptr = NULL;
  }

struct req_line_list *append_attr_value(the_req_ptr, the_val_ptr)

struct req_line_list      *the_req_ptr;
struct ent_value *the_val_ptr;
  {
    struct req_line_list    *req_ptr,
                            *next_req_ptr,
                            *last_req_ptr,
                            *req_line_list_alloc();
    struct ent_value        *val_ptr;

#ifdef EnExFlag
    printf("Enter append_attr_value\n");
#endif

    /* duplicates current req_line for multiple values */
    req_ptr = the_req_ptr;
    while (req_ptr != NULL)
      {
       next_req_ptr = req_ptr->next;
       val_ptr = the_val_ptr;
       while (val_ptr->next != NULL)
         {
          req_ptr->next = req_line_list_alloc();
          strcpy(req_ptr->next->req_line, req_ptr->req_line);
          req_ptr = req_ptr->next;
          val_ptr = val_ptr->next;
```

```
      } /* end while */
     req_ptr->next = next_req_ptr;
     req_ptr = next_req_ptr;
    }

   req_ptr = the_req_ptr;
   while (req_ptr != NULL)
    {
     val_ptr = the_val_ptr;
     while (val_ptr != NULL)
      {
       strcat(req_ptr->req_line, val_ptr->ev_value);
       last_req_ptr = req_ptr;
       req_ptr = req_ptr->next;
       val_ptr = val_ptr->next;
      } /* end while */
    }

#ifdef EnExFlag
   printf("Exit append_attr_value\n");
#endif

   return(last_req_ptr);
  } /* end append_attr_value */


get_base_fun_type(fptr)

struct  function_node  *fptr;
 {
  char   fun_type;

#ifdef EnExFlag
   printf("Enter get_base_fun_type\n");
#endif

   switch (fptr->fn_type)
    {
        case 'i' :
        case 'f' :
        case 's' : fun_type = fptr->fn_type;
                   break;
        case 'b' : fun_type = 'i';
                   break;
        case 'e' : if (fptr -> fn_entptr != NULL ||
                       fptr -> fn_subptr != NULL)
                       fun_type = 'E';
                   else if (fptr -> fn_nonentptr != NULL)
                       fun_type = fptr->fn_nonentptr-> enn_type;
                   else if (fptr -> fn_nonsubptr != NULL)
                       fun_type = fptr -> fn_nonsubptr -> snn_type;
```

```
                else if (fptr -> fn_nonderptr != NULL)
                   fun_type = fptr -> fn_nonderptr -> dnn_type;
                break;
    }  /* end switch */

#ifdef EnExFlag
   printf("Exit get_base_fun_type\n");
#endif

   return(fun_type);

 } /* end get_base_fun_type */


put_dki_ent_value_list(type, str_in)

int  type;
char *str_in:

 {
   struct ent_value    *ev_ptr:
   int                  i;

   if (kms_ptr->dki_evl_ptr.num_values == 0)
        kms_ptr->dki_evl_ptr.type = type;

   ev_ptr = ent_value_alloc();
   ev_ptr->ev_value = var_str_alloc(strlen(str_in) + 1);
   if (type == String)
     for (i = 1; i < strlen(str_in); i++)
          if (isupper(str_in[i]))
            str_in[i] = tolower(str_in[i]);
   strcpy(ev_ptr->ev_value. str_in);
   ev_ptr->next = kms_ptr->dki_evl_ptr.ev_ptr;
   kms_ptr->dki_evl_ptr.ev_ptr = ev_ptr;
   kms_ptr->dki_evl_ptr.num_values++;

 }


clean_dki_evl_ptr()

 {
   struct ent_value      *ev_ptr,
                         *next_ev_ptr;

   kms_ptr->dki_evl_ptr.type = ' ';
   kms_ptr->dki_evl_ptr.num_values = 0;
   ev_ptr = kms_ptr->dki_evl_ptr.ev_ptr;
   while (ev_ptr != NULL)
     {
```

```
      next_ev_ptr = ev_ptr->next;
      free(ev_ptr);
      ev_ptr = next_ev_ptr;
     }
    kms_ptr->dki_evl_ptr.ev_ptr = NULL;
  }


struct ent_node   *search_entity(en_name)
char *en_name;

  {
    int found = FALSE;
    struct ent_node       *entptr;

    entptr =
     cuser_dap_ptr->ui_li_type.li_dap.dpi_curr_db.cdi_db.dn_fun->fdn_entptr;
    while ((entptr != NULL) && (!found))
     {
        if (strcmp(entptr->en_name. en_name) == FALSE)
          found = TRUE;
        else
          entptr = entptr->en_next_ent;
     }

    return(entptr);
  }



struct gen_sub_node   *search_gensub(gsn_name)
char *gsn_name;

  {
    int found = FALSE;
    struct gen_sub_node  *subptr;

    subptr =
     cuser_dap_ptr->ui_li_type.li_dap.dpi_curr_db.cdi_db.dn_fun->fdn_subptr;
    while ((subptr != NULL) && (!found))
     {
        if (strcmp(subptr->gsn_name, gsn_name) == FALSE)
          found = TRUE;
        else
          subptr = subptr->gsn_next_genptr;
     }

    return(subptr);
  }
```

```
struct function_node   *search_funct(indexed_comp_ptr, entptr, subptr)
struct indexed_component  *indexed_comp_ptr;
struct ent_node           *entptr;
struct gen_sub_node       *subptr;

 {
  int found;
  struct function_node  *funct_ptr;
  struct ent_node_list  *enlptr;
  struct sub_node_list  *snlptr;
  struct ent_node               *dummy_entptr = NULL;
  struct gen_sub_node           *dummy_subptr = NULL;

  if (entptr != NULL)
    {
       funct_ptr = entptr->en_ftnptr;
       found = FALSE;
       while ((funct_ptr != NULL) && (!found))
        {
         if (strcmp(funct_ptr->fn_name, indexed_comp_ptr->name_id) == FALSE)
          {
           strcpy(indexed_comp_ptr->parent_name, entptr->en_name);
           found = TRUE;
          }
         else
           funct_ptr = funct_ptr->next;
        }
    }
  else
    {
       funct_ptr = subptr->gsn_ftnptr;
       found = FALSE;
       while ((funct_ptr != NULL) && (!found))
        {
         if (strcmp(funct_ptr->fn_name, indexed_comp_ptr->name_id) == FALSE)
          {
           strcpy(indexed_comp_ptr->parent_name, subptr->gsn_name);
           found = TRUE;
          }
         else
           funct_ptr = funct_ptr->next;
        }

       enlptr = subptr->gsn_entptr;
       while ((!found) && (enlptr != NULL))
        {
         funct_ptr = search_funct(indexed_comp_ptr, enlptr->entptr,
                                                          dummy_subptr);
         if (funct_ptr != NULL)
           found = TRUE;
         else
```

```
            enlptr = enlptr->next;
          }

        snlptr = subptr->gsn_subptr;
        while ((!found) && (snlptr != NULL))
          {
          funct_ptr = search_funct(indexed_comp_ptr. dummy_entptr,
                                                              snlptr->subptr);
          if (funct_ptr != NULL)
            found = TRUE;
          else
            snlptr = snlptr->next;
          }
      }

  return(funct_ptr);
}
```

APPENDIX D

THE DAPLEX GRAMMAR RULES

```
<statement> ::=
        <ddl-statement>
    |   <dml-statement>

<ddl-statement> ::=
        DATABASE <name-id> IS <declarative-item-list> <end-database> SEMICOLON

<end-database> ::=
        END
    |   END <name-id>

<declarative-item-list> ::=
        <declarative-item>
    |   <declarative-item-list> <declarative-item>

<declarative-item> ::=
        <declaration>
    |   <overlap-rule>
    |   <uniqueness-rule>

<overlap-rule> ::= OVERLAP <name1-list> WITH <name1-list> SEMICOLON

<uniqueness-rule> ::= UNIQUE <id-list> WITHIN <name-id> SEMICOLON

<declaration> ::=
        <number-declaration>
    |   TYPE <new-id> <type-declaration>
    |   SUBTYPE <new-id> <subtype-declaration>

<number-declaration> ::=
        <new-id-list> COLON CONSTANT ASSIGN <simple-const> SEMICOLON

<simple-const> ::=
        INTEGER-LITERAL
        FLOAT-LITERAL
        CHARACTER-STRING

<type-declaration> ::=
        IS <type-definition> SEMICOLON
    |   IS <entity-type-definition> SEMICOLON
    |   <incomplete-type-declaration>

<incomplete-type-declaration> ::= SEMICOLON

<type-definition> ::=
        <enumeration-type-definition>
    |   <integer-range>
```

```
        <float-range>
    |   <derived-type-definition>

<enumeration-type-definition> ::= LP <enumeration-literal-list> RP

<enumeration-literal-list> ::=
        <enumeration-literal>
    |   <enumeration-literal-list> COMMA <enumeration-literal>


<enumeration-literal> ::= IDENTIFIER

<integer-range> ::= RANGE <int-range>

<int-range> ::= INTEGER-LITERAL ELIPSES INTEGER-LITERAL

<float-range> ::= RANGE FLOAT-LITERAL ELIPSES FLOAT-LITERAL

<derived-type-definition> ::= NEW <name-id> <derived-range>

<derived-range> ::=
        <integer-range>
    |   <float-range>

<entity-type-definition> ::=
        ENTITY <entity-component-declaration-list> END ENTITY
    |   ENTITY END ENTITY

<entity-component-declaration-list> ::=
        <entity-component-declaration>
    |   <entity-component-declaration-list> <entity-component-declaration>

<entity-component-declaration> ::= <name1-list> COLON <function-type-declaration>

<function-type-declaration> ::=
        <function-type> <end-scalar-function>
    |   <set-type-definition> SEMICOLON

<set-type-definition> ::= SET OF <function-type>

<end-scalar-function> ::=
        SEMICOLON
    |   ASSIGN <default-value> SEMICOLON

<default-value> ::=
        INTEGER-LITERAL
    |   FLOAT-LITERAL
    |   CHARACTER-STRING
    |   <boolean-value>

<boolean-value> ::=
```

```
        TRUE
    |   FALSE

<function-type> ::=
        <type-mark> <constraint>
    |   STRING LP <string-range> RP

<string-range> ::=
        <int-range>
        INTEGER-LITERAL

<type-mark> ::=
        <name-id>
        FLOAT
    |   INTEGER
    |   BOOLEAN

<constraint> ::=
        /* empty */
        <range-constraint>
        <null-constraint>

<range-constraint> ::=
        <integer-range>
        <float-range>
        <enumeration-range>

<enumeration-range> ::= RANGE IDENTIFIER ELIPSES IDENTIFIER

<null-constraint> ::=
        WITHNULL
    |   WITHOUTNULL

<subtype-declaration> ::=
        IS <complete-subtype> SEMICOLON
    |   <incomplete-subtype-declaration>

<complete-subtype> ::=
        <name-id subtype-definition>
    |   <id-list> <entity-type-definition>
    |   STRING LP <string-range> RP

<subtype-definition> ::=
        RANGE <enumeration-literal> ELIPSES <enumeration-literal>
    |   <integer-range>
    |   <float-range>
    |   /* empty */

<incomplete-subtype-declaration> ::= SEMICOLON

<name-id> ::= IDENTIFIER
```

```
<new-id> ::= <name-id>

<new-id-list> ::=
        <new-id>
    |   <new-id-list> COMMA <new-id>

<id-list> ::=
        <name-id>
    |   <id-list> COMMA <name-id>

<name1> ::= <name-id>

<name1-list> ::=
        <name1>
        <name1-list> COMMA <name1>

<dml-statement> ::=
        <create-statement>
    |   <destroy-statement>
        <move-statement>
        <loop-statement>

<dml-statement2> ::=
        <assignment-statement>
        <include-statement>
    |   <exclude-statement>
    |   <destroy-statement>
    |   <move-statement>
    |   <procedure-call>

<create-statement> ::= CREATE NEW <create-part> SEMICOLON

<create-part> ::= <name1-list> <named-aggregate>

<named-aggregate> ::=
        LP <component-association-list> RP
    |   /* empty */

<component-association-list> ::=
        <component-association>
    |   <component-association-list> COMMA <component-association>

<component-association> ::= IDENTIFIER IMPLY <dap-expr>

<dap-expr> ::=
        <relation>
    |   <rel-and-list>
    |   <rel-or-list>

<rel-and-list> ::=
        <relation> AND <relation>
```

```
    |   <rel-and-list> AND <relation>

<rel-or-list> ::=
        <relation> OR <relation>
    |   <rel-or-list> OR <relation>

<relation> ::=
        <simple-expr>
    |   <simple-expr> <relational-operator> <simple-expr>
    |   <simple-expr> <in-op> <dap-range>
    |   <simple-expr> <in-op> <name1>

<simple-expr> ::=
        <literal>
    |   <set-constructor>
    |   <indexed-component>
    |   <function-application>

<function-application> ::= <function-name> LP <expr-types> RP

<expr-types> ::=
        <name1>
        <set-constructor>
        <indexed-component>

<indexed-component> ::=
        <dml-id-name> LP <dml-id-name> RP
    |   <dml-id-name> LP <indexed-component> RP

<set-constructor> ::= LCB <end-set-construct>

<end-set-constructor> ::=
        <basic-expr-list> RCB
    |   <simple-expr> IN <set-constructor-part> WHERE <dap-expr> RCB
    |   RCB

<set-constructor-part> ::=
        <name1>
    |   <dap-range>

<basic-expr> ::=
        <literal>
    |   <indexed-component>
    |   <function-application>

<basic-expr-list> ::=
        <basic-expr>
    |   <basic-expr-list> COMMA <basic-expr>

<domain> ::=
        <loop-expr>
```

```
    |   <loop-expr> WHERE <dap-expr>

<loop-expr> ::=
        <indexed-component>
    |   IDENTIFIER

<exclude-statement> ::= EXCLUDE <dap-expr> FROM <indexed-component> SEMICOLON

<destroy-statement> ::= DESTROY <dap-expr> SEMICOLON

<assignment-statement> ::= <indexed-component> ASSIGN <dap-expr>

<include-statement> ::= INCLUDE <dap-expr> INTO <indexed-component> SEMICOLON

<move-statement> ::= MOVE <dap-expr> <move-statement2> SEMICOLON

<move-statement2> ::=
        <move-from>
    |   <move-to>
    |   <move-from> <move-to>

<move-from> ::= FROM <name1-list>

<move-to> ::= INTO <create-part>

<procedure-call> ::= <procedure-name> LP <basic-expr-list> RP SEMICOLON

<procedure-name> ::=
        PRINT
    |   PRINT-LINE

<loop-statement> ::=
        <real-loop> SEMICOLON
    |   IDENTIFIER COLON <real-loop> IDENTIFIER SEMICOLON
    |   IDENTIFIER COLON <real-loop> SEMICOLON

<real-loop> ::= <iteration-clause> <basic-loop> <end-loop>

<iteration-clause> ::=
        <iteration-body>
    |   <iteration-body> BY <order-component-list>

<iteration-body> ::= <for-clause> IDENTIFIER IN <domain>

<for-clause> ::=
        FOR
    |   FOR EACH

<order-component-list> ::=
        <order-component>
    |   <order-component-list> COMMA <order-component>
```

```
<order-component> ::=
        <indexed-component>
    |   <sort-order> <indexed-component>

<sort-order> ::=
        ASCENDING
    |   DESCENDING

<basic-loop> ::=
        <sequence-of-statements>
    |   LOOP <sequence-of-statements>

<sequence-of-statements> ::=
        <dml-statement2>
    |   <sequence-of-statements> <dml-statement2>

<end-loop> ::=
        END
    |   END LOOP

<dml-id-name> ::= IDENTIFIER

<relational-operator> ::=
        EQ
    |   NE
    |   LT
    |   LE
    |   GT
    |   GE

<in-op> ::=
        IN
    |   NOT IN

<dap-range> ::=
        <integer-range>
    |   <float-range>

<literal> ::=
        CHARACTER-STRING
    |   NULL
    |   TRUE
    |   FALSE

<function-name> ::=
        COUNT
    |   SUM
    |   AVG
    |   MIN
    |   MAX
```

LIST OF REFERENCES

1. Demurjian, S. A., *The Multi-Lingual Database System - A Paradigm and Test-Bed for the Investigation of Data-Model Transformations, Data-Language Translations and Data-Model Semantics*, Ph. D. Dissertation, The Ohio State University, Columbus, Ohio, December 1986.
2. Weishar, D. J., *The Design and Analysis of a Complete Relational Hierarchical Interface for a Multi-Backend Database System*, Master's Thesis, Naval Postgraduate School, Monterey, California, June 1984.
3. Macy, G., *Design and Analysis of an SQL Interface for a Multi-Backend Database System*, Master's Thesis, Naval Postgraduate School, Monterey, California, March 1984.
4. Wortherly, C. R., *Design and Analysis of a Complete Network Interface for a Multi-Backend Database System*, Master's Thesis, Naval Postgraduate School, Monterey, California, December 1985.
5. Rollins, R., *Design and Analysis of a Complete Relational Interface for a Multi-Backend Database System*, Master's Thesis, Naval Postgraduate School, Monterey, California, June 1984.
6. Goisman, P. L., *Design and Analysis of a Complete Entity-Relationship Interface for the Multi-Backend Database System*, Master's Thesis, Naval Postgraduate School, Monterey, California, December 1985.
7. Naval Postgraduate School, Monterey, California, Technical Report, NPS-52-85-009, *Design Analysis and Performance Evaluation Methodologies for Database Computers*, by Demurjian, S. A., et al., June 1985.
8. Benson, T. P. and Wentz, G. L., *The Design and Implementation of a Hierarchical Interface for the Multi-Lingual Database System*, Master's Thesis, Naval Postgraduate School, Monterey, California, June 1985.
9. Kloepping, G. R. and Mack, J. F., *The Design and Implementation of a Relational Interface for the Multi-Lingual Database System*, Master's Thesis, Naval Postgraduate School, Monterey, California, June 1985.
10. Emdi, B., *The Design and Implementation of a Network [Codasyl-DML] Interface for the Multi-Lingual Database System*, Master's Thesis, Naval Postgraduate School, Monterey, California, December 1985.
11. Anthony, J. A. and Billings, A. J., *The Implementation of an Entity-Relationship [Daplex] Interface for the Multi-Lingual Database System*, Master's Thesis, Naval Postgraduate School, Monterey, California, December 1985.
12. Hsiao, D. K., and Harary, F., "A Formal System for Information Retrieval from Files," *Communications of the ACM*, Vol. 13, No. 2, pp. 67-73, February 1970, also in *Corrigenda*, Vol. 13, No. 4, April 1970.

13. Wong. E.. and Chiang. T. C.. "Canonical Structure in Attribute Based File Organization." *Communications of the ACM*, Vol. 14, No. 9, pp. 593-597, September 1971.

14. Rothnie. J. B. Jr., "Attribute Based File Organization in a Paged Memory Environment," *Communications of the ACM*, Vol. 17, No. 2, pp. 63-69, February 1974.

15. The Ohio State University, Columbus. Ohio, Technical Report No. OSU-CISRC-TR-77-7, *DBC Software Requirements for Supporting Relational Databases*, by J. Banerjee and D. K. Hsiao, November 1977.

16. Shipman, D. W., "The Functional Data Model and the Data Language Daplex." *ACM Transaction on Database Systems*, Vol. 6, No. 1, March 1981.

17. Gray, P. M. D., *Logic, Algebra and Databases*, Halsted Press, 1984.

18. Fox. S.. et al.. *Daplex User's Manual, CCA-84-01*, Computer Corporation of America. Four Cambridge Center. Cambridge. Massachusetts. June 1984.

19. Boehm. B. W.. *Software Engineering Economics*. Prentice-Hall, pp. 14-46. 1981.

20. The Ohio State University, Columbus. Ohio. Technical Report No. OSU-CISRC-TR-82-1. *The Implementation of a Multi Backend Database System (MDBS): Part I - Software Engineering Strategies and Efforts Towards a Prototype MDBS*, by D. S. Kerr et al, January 1982.

21. Kernighan. B. W. and Ritchie, D. M., *The C Programming Language*, Prentice-Hall, 1978.

22. Holste, S. T., *The Implementation of a Multi-Lingual Database System -- Multi-Backend Database System Interface* Master's Thesis, Naval Postgraduate School. Monterey, California, June 1986.

23. Johnson, S. C., *Yacc: Yet Another Compiler-Compiler*, Bell Laboratories, Murray Hill, New Jersey, July 1978.

24. Lesk, M. E. and Schmidt, E., *Lex - A Lexical Analyzer Generator*, Bell Laboratories, Murray Hill. New Jersey, July 1978.

25. *Adaplex BNF*, Computer Corporation of America, Four Cambridge Center, Cambridge, Massachusetts, May 1985.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria. Virginia 22304-6145	2
2.	Library. Code 0142 Naval Postgraduate School Monterey. California 93943-5002	2
3.	Department Chairman. Code 52 Department of Computer Science Naval Postgraduate School Monterey. California 93943-5000	2
4.	Curricular Officer. Code 37 Computer Technology Naval Postgraduate School Monterey, California 93943-5000	1
5.	Professor David K. Hsiao, Code 52Hq Computer Science Department Naval Postgraduate School Monterey, California 93943-5000	2
6.	Steven A. Demurjian, Code 52 Computer Science Department Naval Postgraduate School Monterey, California 93943-5000	2
7.	Beng Hock Lim 507, Bedok North Ave 3 #10-347, Singapore 1646 Republic of Singapore	3

Thesis
L63855 Lim
c.1 The implementation of a functional/daplex interface for the multi-lingual database system.