NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

THE DESIGN AND IMPLEMENTATION OF A COMPILER FOR THE OBJECT-ORIENTED DATA DEFINITION LANGUAGE

by

Luis M. Ramirez and Recep Tan
September 1995

Thesis Co-Advisors: David K. Hsiao
C. Thomas Wu

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)

2. REPORT DATE
Sept 1995

3. REPORT TYPE AND DATES COVERED
Master's Thesis

4. TITLE AND SUBTITLE
THE DESIGN AND IMPLEMENTATION OF A COMPILER FOR THE OBJECT-ORIENTED DATA DEFINITION LANGUAGE

5. FUNDING NUMBERS

6. AUTHOR(S)
Ramirez, Luis M.,
Tan, Recep.

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/ MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING/ MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Classic data models such as the Relational and Hierarchical do not have capabilities to handle both of the object-oriented relationships, inheritance and covering. Therefore, the problem addressed by this work is to design and implement a completely new data model that embodies the object-oriented paradigm. With such an object-oriented data model (O-ODM), the direct modelling of a variety of database applications becomes possible.

Database research at the Naval Postgraduate School has produced a Multimodeland Multilingual Database System called M^2DBS. M^2DBS currently supports all the classic database data models as well as a newly developed O-ODM. The approach taken is to first develop and build an entirely self-sufficient O-ODDL Compiler. Then, incorporate this compiler into the Kernel Mapping System (KMS) of the M^2DBS.

The results of this thesis is a compiler for the object-oriented data definition language (O-ODDL) of the O-ODM. This O-ODDL compiler takes an O-ODM database specification as input and does an automatic translation into the data format recognized by the M^2DBS.

14. SUBJECT TERMS
Object-Oriented data model, Object-Oriented data definition language compiler, Multimodel/Multilingual Database Management System

15. NUMBER OF PAGES
115

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

Approved for public release; distribution is unlimited

THE DESIGN AND IMPLEMENTATION OF A COMPILER FOR THE OBJECT-ORIENTED DATA DEFINITION LANGUAGE

Luis M. Ramirez
Lieutenant, United States Navy
B.S., University Of California at Los Angeles, 1986

and

Recep Tan
Lieutenant Junior Grade, Turkish Navy
B.S., Turkish Naval Academy at Istanbul, 1989

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September 1995

Authors: ______________________________
Luis M. Ramirez

____
Recep Tan

Approved by: ______________________________
David K. Hsiao, Thesis Co-Advisor

____
C. Thomas Wu, Thesis Co-Advisor

____
Ted Lewis, Chairman,
Department of Computer Science

ABSTRACT

Classic data models such as the Relational and Hierarchical do not have capabilities to handle both of the object-oriented relationships, inheritance and covering. Therefore, the problem addressed by this work is to design and implement a completely new data model that embodies the object-oriented paradigm. With such an object-oriented data model (O-ODM), the direct modelling of a variety of database applications becomes possible.

Database research at the Naval Postgraduate School has produced a Multimodel and Multilingual Database System called M^2DBS. M^2DBS currently supports all the classic database data models as well as a newly developed O-ODM. The approach taken is to first develop and build an entirely self-sufficient O-ODDL Compiler. Then, incorporate this compiler into the Kernel Mapping System (KMS) of the M^2DBS.

The results of this thesis is a compiler for the object-oriented data definition language (O-ODDL) of the O-ODM. This O-ODDL compiler takes an O-ODM database specification as input and does an automatic translation into the data format recognized by the M^2DBS.

TABLE OF CONTENTS

LIST OF FIGURES

I. INTRODUCTION

A. THE BACKGROUND

The conventional design approach of database systems has been to produce a mono-model and mono-lingual system. Such a system is one where the user sees and utilizes the database system with a specific data model and its model-based data language. Some examples of these database systems are:

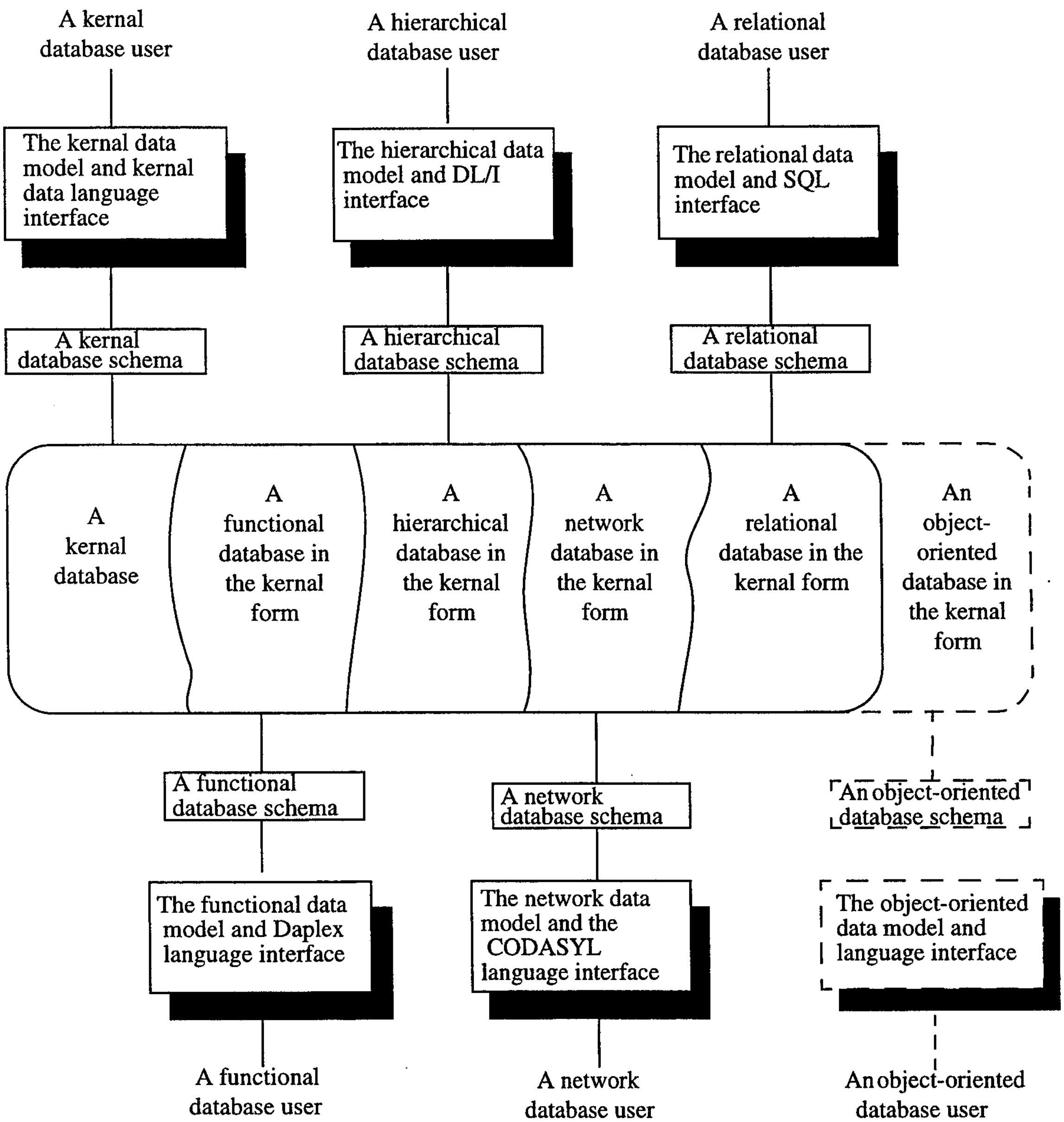
- IBM's SQL/Data System which supports the relational model and IBM's Structured English Query Language (SGL)
- IBM's Information Management System (IMS) which supports the hierarchical model and IBM's Data language I (DL/I)
- Univac's CODASYL-DML/Data System which supports the network model and Univac's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

These mono-model and mono-lingual database systems are all designed to meet specific application requirements. For example, the relational database system is designed for keeping records, the hierarchical database system is designed for tracking the product assembly; the network database system is designed for controlling inventories, and the functional database system is designed for making inferences. In order to accommodate varied applications, an organization is forced to support multiple database systems, i.e., all these mono-model and mono-lingual type database systems. But, all these application-specific database systems have one severe drawback. They lack the ability to share data among themselves. Our research effort to overcome this limitation has been to introduce a new and unconventional approach to the design and implementation of a database system, known as the multi-model and multi-lingual database system for sharing data among heterogeneous databases [Ref. 1].

The Multi-model and Multi-lingual Database Management Systems (M^2DBMS), at the Naval Postgraduate School's Laboratory for Database Systems Research, is a single database system that can execute many transactions written in different data languages and support many databases structured on different data models. M^2DBMS supports the aforementioned data models with a single data model, the Attribute-Based Data Model (ABDM), and the aforementioned data languages with a single data language, the Attribute-Based Data Language (ABDL) [Ref. 2]. We have developed an attribute-based database system which supports hierarchical, relational, network, and functional databases, and runs transactions in DL/I, SQL, CODASYL-DML, and Daplex on their respective databases. However, in order to accommodate new application requirements of the future, an additional goal is to support a new pair of data model and data language, known as Object-Oriented Data Model, and Object-Oriented Data Language.

B. THE MOTIVATION

The work completed for this thesis is part of a large research effort to design and build a new data-model-and-data-language support in M^2DBMS, i.e., Object-Oriented-Model-and-Object-Oriented-Language Interface. This thesis is focused on the *object-oriented-Data-Model Interface*. See Figure 1 of the interface in the context with other interfaces. This new interface supports a database with Object-Oriented constructs such as objects, classes, inheritances, and coverings. For a detailed description of these constructs and other related constructs, see references [Ref. 3]. In this thesis, the design and implementation of a compiler which converts a database specified in the Object-Oriented Data Definition Language (O-ODDL) into an equivalent database in the attribute-based data model (ABDM) are elaborated. The attribute-based database is supported in the attribute-based database system (ABDBMS).



(1) The term "kernal" means "attribute-based" in this figure.
(2) Solid lines characterize existing software and data. Dashed lines characterize present research efforts.

Figure 1. The Multimodel and Multilingual Database Management System (M^2DBMS).

We utilize compiler-writing tools such as Lex and YACC for the implementation of the O-ODDL Compiler. First, the O-ODDL Compiler scans tokens of an O-ODDL specification of an Object-Oriented database, and rejects unacceptable ones. Next, the parser of the O-ODDL Compiler uses the scanned and accepted tokens to verify the syntactic and semantic correctness of O-ODDL statements. Concurrent to the parsing, dynamic storage structures are filled with data for the Object-Oriented database that will be used in the production of equivalent ABDM (called kernal informally) constructs and the attribute-based (kernal) database. The final step is to incorporate the compiler into the existing M^2DBMS.

C. THE ORGANIZATION OF THIS THESIS

The remainder of this thesis is organized into eight chapters and eight appendices. In Chapter II, we present a summary of project goals and how the work for this thesis fits into the other project efforts. In Chapter III, we present the background material: an overview of the source data model and language, i.e., the O-ODM and O-ODDL, and the target data model and language, ABDM and ABDL. In Chapter IV, we present the design of three compiler components: the scanner, the parser, and the code generator. Along with an overview of the UNIX compiler tools, Lex and YACC. In Chapter V, we present the O-ODDL and its subsequent lexical analysis using Lex. In Chapter VI, we present the grammar and production rules of the O-ODDL and their syntactical analysis (i.e., parsing) and productions using YACC. In Chapter VII, we describe the compiler output: the descriptor file, the template file, and the data dictionary, which constitute the ABDM equivalent of an O-ODDL specification. In Chapter VIII, we present a summary of the logic of the M^2DBMS, and the incorporation of the newly completed O-ODDL Compiler into M^2DBMS. Finally, in Chapter IX, we make concluding remarks on accomplishments and limitations.

Of the appendices, Appendix A contains a sample Object-Oriented database specification. We reference to this specification throughout the thesis; Appendix B has the listing of the O-ODDL scanner program written in the Lex format; Appendix C has the listing of the basic O-ODDL parser program in the YACC format; Appendix D contains the data structures used for the object-oriented-data-model interface; Appendix E contains a complete tabular listing of the Data Dictionary that corresponds to the sample database given in Appendix A; Appendix F has the listing of the final O-ODDL parser program in the YACC format, which includes a generator for the target language code; Appendix G contains sample output files produced by the compiler; Finally, Appendix H is the user manual for the O-ODDL Compiler.

II. THE IMPLEMENTATION PROCESS

The overall goal of the project, in which this thesis research is a part, is to design, implement, and add an entirely new object-oriented-data-model-and-language-interface to the M^2DBMS. Since there is an entirely new data model in the interface, there exists no specification for the object-oriented data-modeled database given. So, the features and requirements for such a database are defined first.

A. CONSTRUCTS FOR THE IMPLEMENTATION

The specifications for our object-oriented database are based on features and constructs borrowed mostly from object-oriented programming languages. The following is brief overview of concepts associated with the object-oriented paradigm. Refer to [Ref. 3] for a more detailed discussion.

1. Object-Oriented Constructs

The object-oriented constructs, for a data model must incorporate at the minimum: attributes, methods, objects, object identifiers, object classes, inheritance, and covering.

a. Objects

An *object* is the most fundamental or basic construct in the object-oriented data model. Objects are simply collections of data. More specifically, each collection, i.e., an object, consists of the values of certain *attributes* and names of known *methods*. An object is said to be an *instance* of a class which is defined in paragraph *c*.

b. Object Identifiers

Each object is assigned a system-defined *object identifier* (OID). All OIDs are distinguishable and unique. With these OIDs, the sharing of objects is possible. This sharing of objects has two primary benefits. The first benefit is that the actual physical storage requirements of the database is reduced. Second, the updating and integrity problem of traditional databases is reduced due to the absence of redundant data.

c. Classes

A *class* is a grouping of objects which share common attributes, methods, or both. A class is defined by one or two parts, a set of attributes and a, possibly empty, set of methods. The set of attributes defines the data that can be stored in a class and the data are termed objects. The set of methods defines the operations permitted on objects of the class. A class contains no data, but rather, all data held in a class are in its instances, i.e., objects. In short, a class merely serves as a template with which an instance of a class may be created.

d. Inheritance

Inheritance establishes a relationship of two or more classes. We say that of two classes, A and B, where B inherit A, if class B has all the properties of class A. In this case, class A is said to be the *superclass* of the *subclass* B. A superclass can also be referred to as a *generalization* of all its subclasses, because all the properties of the superclass form a common subset of the properties in all the subclasses. Conversely, a subclass can be referred to as a *specialization*, because it not only contains the common properties of a superclass, but it also possess properties which are unique to it alone. In short, the *Inheritance* class relationship is where a subclass has all the attributes and methods of its superclass. And such a subclass can also have additional attributes and methods that are not found in the superclass.

e. Covering

Covering is another relationship of two classes in the object-oriented data model. Two classes are said to have the covering relationship, if every object of one class, A, is mapped to, or corresponds to, a subset of objects of the second class, B. In this instance, class A is said to *cover* class B. Class A is referred to as the *cover class* and class B is referred to as the *member class* [Ref. 4].

B. THE IMPLEMENTATION STRATEGY

The Object-Oriented Data Model (O-ODM) is the foundation of a new object-oriented data language. The design and specification of this new language are the first step in the research project which can be found in [Ref. 3]. After the data requirements and construct representations for the object-oriented data language have been defined, the actual design and implementation of the new O-ODM-based data language compiler can begin.

The design and implementation of this research project is divided into two areas: the design and implementation of a compiler for the Object-Oriented Data Definition Language (O-ODDL), and another compiler for the Object-Oriented Data Manipulation Language (O-ODML). Together, the O-ODDL and O-ODML form the object-oriented data language of the object-oriented data model. In this thesis, we focus on the design and implementation of the O-ODDL Compiler. The design and the implementation of the O-ODML Compiler can be found in [Ref. 5] and [Ref 6].

III. THE COMPILER SOURCE AND TARGET DATA LANGUAGES

The utility of the Object-Oriented database model is measured by its ability to conceptually define and represent real-world objects. These objects must then have certain constraints on them and their relationship to other objects. It is these conceptual representations that are specifically defined by the Object-Oriented Data Language (O-ODL). Refer to [Ref. 3] for a thorough discussion of the O-ODL design and development.

A. FORMATS AND SPECIFICATIONS OF THE SOURCE

The underlying constructs used to define an object-oriented database have been discussed in the previous chapter, and they included the following: objects, classes, inheritances, and coverings. The most basic of these is the object. An object can be any entity in an application. Once the application's objects are identified, they may be combined into classes of similar objects.

1. The Specification of an Object Class

In Figure 2, we depict the generic structure used for a definition of an object-oriented class.

Class *Class_name* {
attribute_type *attribute_name*$_1$;
⋮
attribute_type *attribute_name*$_x$;
return_type *method_name*$_1$;
⋮
return_type *method_name*$_y$;
};

Figure 2. The definition structure of a generic O-ODDL class.

The rudimentary structure for the definition of an O-ODDL class is modeled after class structures in the C++ programming language [Ref. 7]. The *Class_name* is the name assigned to a particular class of similar objects. The *attribute_type* is the declared type for the corresponding *attribute_name*. Valid attribute types are: char for character, int for integer, char_string for a character string, and, lastly, class_name for another class. The *attribute_name* are the names given to the variables which make up the specific values of a class. The concept of class methods and corresponding structures, are not implemented in this research project. But are only depicted to demonstrate where such structures could be added in future research efforts.

2. The Specification of Object-Oriented Constructs in a Sample Database

A sample object-oriented database, FACSTU, is used as an example throughout this thesis. We use it to illustrate how class relationships are implemented. The O-ODDL handles four class relationships: inheritance, covering, set_of, and inverse_of. All of these relationships can be illustrated in a class hierarchy which is a collection of similar objects with these relationships. In Figure 3, the FACSTU database diagram represents a class hierarchy. Classes of similar objects are formed into the class hierarchy to represent their class relationships and respective constraints. The features in which the class hierarchy embodies are class generalizations and class specializations, and their inheritances, and other specific relationships such as the covering, the set_of, and the inverse_of.

The respective generalizations and specializations of various class objects are used to construct the class hierarchy. Referring to Figure 3, the generalized class, which can also be thought of as the superclass, is the Person class. This superclass is then a generalization of the two subclasses, Faculty and Student. And the two subclasses are in turn specializations of the superclass. All common properties of the subclasses are maintained by the superclass. In this case, the common attributes, such as, *fname*, *street*, and *zipcode*, are stored in the superclass Person.

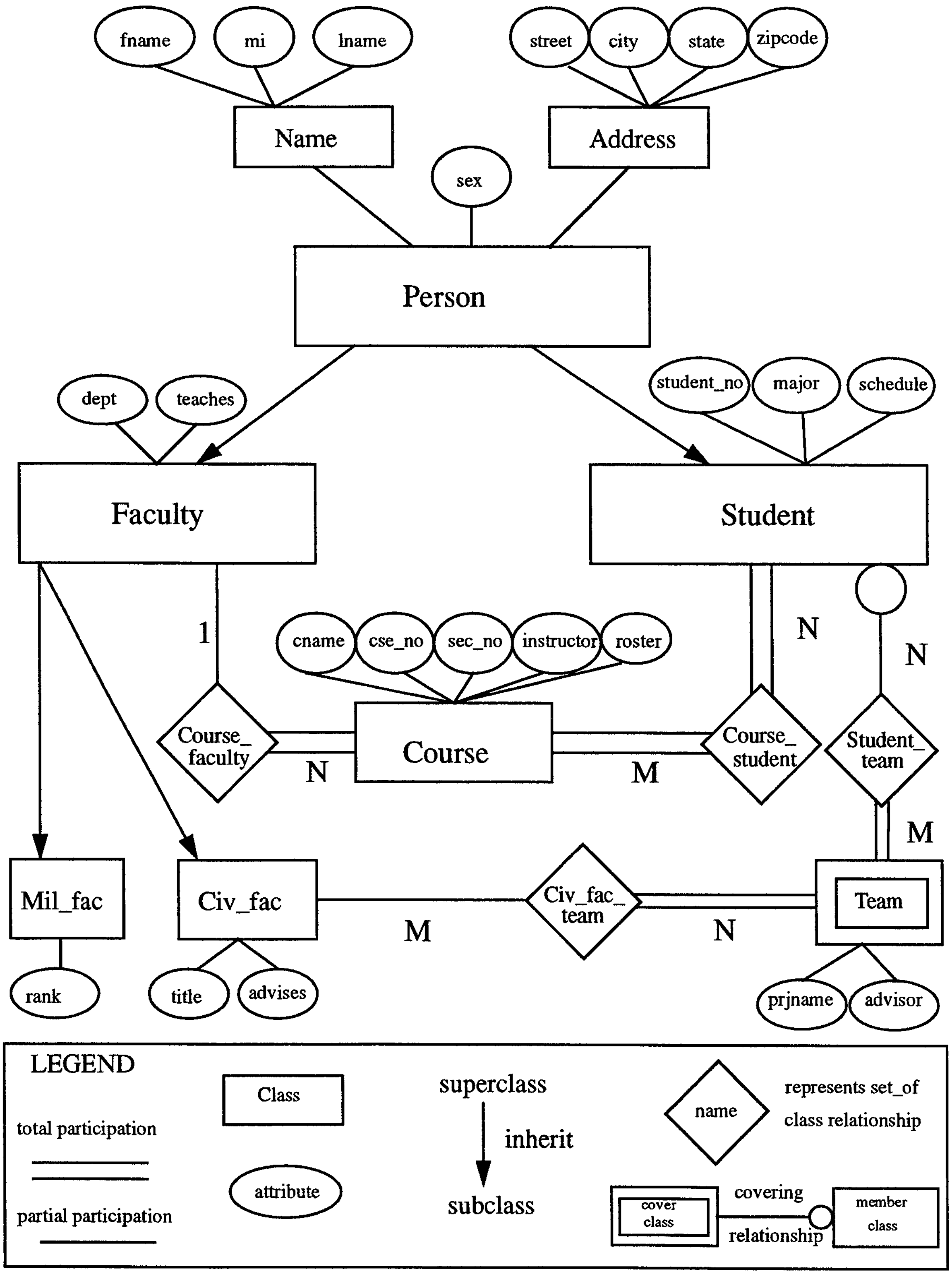


Figure 3. The FACSTU Database.

Class inheritance, or simply inheritance, is the linking element that more specifically define the class hierarchical composition. Inheritance is a further refinement of generalizations and specializations, because a specialized subclass inherits the properties, i.e., the attributes and methods, of its superclass. In our sample in Figure 3, the Student subclass inherits all the attributes and methods from superclass Person, and is therefore pointed by an arrow.

As stated in the previous chapter, the covering relationship is a property that allows every object of a specific class, the cover class, to map to a corresponding subset of objects of another class, the member class. In Figure 3, the class Team is a cover class, the class Student is a member class, and the covering relationship is delineated by a line with a circle, where the circle is at the member class of the covering. With this example of the covering relationship, we can say that every student belongs to one or more teams.

The set_of relationship is used to build 1:N and M:N class relationships among objects. More precisely, the set_of relationship establishes a set relationship between classes. In Figure 3, we depict graphically this relationship. But in reality, a user would not actually draw these structures, because its specification is done by the database designer and its implementation is done automatically by the O-ODDL Compiler. We add graphical depictions to illustrate how the set_of relationship ties in with a class hierarchy.

The last relationship, inverse_of, is the compliment of the set_of relationship. The class hierarchy with all the aforementioned constructs are required in the implementation of the O-ODM. In the M^2DBMS, these constructs are supported by means of an object-oriented schema.

3. The Role of an Object-Oriented Schema

The *object-oriented schema* is the specification of object-oriented data in a database. Additionally, the schema is the means with which all proposed object-oriented constructs in the database can be realized in the M^2DBMS. In Figure 4, there is an example of an object-oriented schema. Refer to Appendix A for a complete listing of the object-oriented schema of the FACSTU database depicted in Figure 3.

```
Class Class_name {
  attribute_type        attribute_name1;
                 :
  attribute_type        attribute_namex;
};
                 :
Class Subclass_name : Inherit Superclass_name {
  attribute_type          attribute_name1;
                 :
  attribute_type          attribute_namex;
  set_of Class_name       attribute_name;  *
};
                 :
Class cover_class_name : Cover member_class_name{
  attribute_type                 attribute_name1;
                 :
  attribute_type                 attribute_namex;
  inverse_of Class_name          attribute_name;  *
};
```

(*) **set_of** and **inverse_of** can be placed into any Class structure

Figure 4. The Generic Object-Oriented Schema format.

4. The Object-Oriented Data Language

In the design and development of our O-ODDL, we focused on two primary considerations. First, the object-oriented data definition language must be easy to understand and use. That is why it is modelled after the very popular C++ language. Second, the object-oriented data definition language must efficiently map into the attribute-based data language that creates the database. We believe that our O-ODDL and its compiler have met these two considerations.

B. THE TARGET DATA LANGUAGE FORMAT AND SPECIFICATION

As stated previously, M^2DBMS supports many databases based on different data models, and their respective data languages. In order to support these different data models and data languages, M^2DBMS has a single pair of data model and data language which serve as the kernal of all data models and data languages. The kernal data model and kernal data language used in M^2DBMS is the attribute-based data model and attribute-based data language (ABDM and ABDL) [Ref. 8]. Therefore, ABDM is the target data model in which our compiler ultimately produces the database specification. More precisely, our O-ODDL Compiler produces an ABDM specification from an O-ODL specification which is written in O-ODDL.

1. The Attribute-Based Data Model (ABDM)

The foundation of ABDM is the *attribute-value pair*. The attribute defines the specific quality or the certain characteristics of the value. An example would look like the following, <fname, John>. Where this attribute-value pair defines fname (an acronym for first name) as the attribute, and the name John as the value for that attribute. A *record body* is the textual information pertanent to a specific record.

We combine many attribute-value pairs and a record body into a set, called a *record.* And a *database* can be thought of as simply a collection of records. But in order for these records to form a database under ABDM, the attribute-value pairs that comprise each record are subject to three constraints: (1) No attribute can be repeated in a record. (2) An attribute can not have more than one value in the record. (3) Every record must have at least one keyword, or *key* for short. Figure 5 consists of an example of a record.

(<TEMP, Name>, <OID, N1>, {<FNAME, John>, <MI, J>, <LNAME, Doe>})

Figure 5. An example of an ABDM Record.

The words enclosed in the angled brackets, <, >, represent attribute-value pairs, for short keywords. 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 records (files). <TEMP, Name> and <OID, N1> in Figure 5 are examples of directory keywords. The directory keyword <OID, N1> represents object identifier, and is implemented because according to the object-oriented construct that every object must be unique and distinguishable from all other objects. In this case, an object is a record, which is assigned a unique object identifier, OID. The curly brackets {, }, enclose the record body. The entire record is enclosed within the paratheses.

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, an attribute value, e.g., (LNAME = Doe). These keyword predicates are used to write queries. A *query* combines keyword predicates in a disjunctive normal form. An example of a query is given in Figure 6. The query will be satisfied by all records of the Name template (TEMP) where the attribute value of FNAME is "John" or the attribute value of LNAME is "Doe". We use parentheses for bracketing conjunctions in a query.

((TEMP = Name) and (FNAME = John)) or

((TEMP = Name) and (LNAME = DOE))

Figure 6. An example of a Query for ABDM data.

2. The Attribute-Based Data Language (ABDL)

The ABDL supports five *primary database operations*: INSERT, DELETE, UPDATE, RETRIEVE, and RETRIEVE-COMMON. A request in the ABDL is specified with a primary operation that has a qualification. A *qualification* specifies the part of the database that a particular operation applies. Two or more requests may be grouped together to form a transaction. Since we need only one primary operation as our target operation, we forgo any discussion of the other four.

The INSERT request inserts a new record into the database. The quantification of an INSERT request is a list of keywords with or without a record body. In Figure 7, there is an example of an INSERT request. This is the only ABDL used by the O-ODDL compiler to generate a database based on ABDM. We do not discuss the other four primary operations here, which can be found in [Ref. 9].

INSERT (<TEMP, Name>, <OID, N2>, <FNAME, Jane>,

<MI, C>, <LNAME, Doe>)

Figure 7. An example INSERT Request.

Our O-ODDL Compiler produces a descriptor file, a template file, and a data dictionary. To create an object-oriented database in M^2DBMS, the descriptor file and template files are used in which INSERTS are embedded. In the following Chapters, we thoroughly discuss the descriptor file, the template file, the data dictionary, and their relationship with attribute-value pairs and INSERT operations.

IV. OVERALL COMPILER DESIGN CONCEPTS

A. COMPILER COMPONENTS

A compiler is simply a program that reads a program written in one language, the source language, and translates it into an equivalent program in another language, the target language. In our case, the source language is the O-ODDL, and the target language is the ABDL.

The compilation process is composed of two components: analysis and synthesis. The analysis part breaks up the source program into constituent pieces and creates an intermediate representation of the source program. The synthesis component constructs the desired target program from the intermediate representation [Ref. 10]. In actuality, the analysis component is composed of two other sub-components: the scanner for lexical analysis, and the parser for syntactic analysis. Figure 8 shows the flow of a language translation through these compiler components. In the following three respective sections, we will elaborate on each of these three major components that comprise our compiler model.

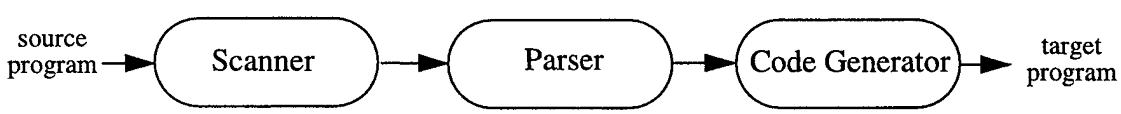


Figure 8. Basic compiler flow diagram.

1. The Scanner

The scanner, or lexical analyzer, is the first phase of a compiler. Its main task is to read the input characters and produce as output a sequence of tokens that the parser uses for syntax analysis. A *token* is a sequence of characters having a collective meaning.

The interaction of the scanner with the parser is summarized in Figure 9. What this figure shows is that the scanner is implemented as a subroutine or a coroutine of the parser. Upon receiving a "get next token" command from the parser, the scanner reads input characters until it can identify the next token. Examples of valid input tokens are: reserved words, symbols, numerical expressions, and identifiers. And these tokens are typically stored in a reference symbol table.

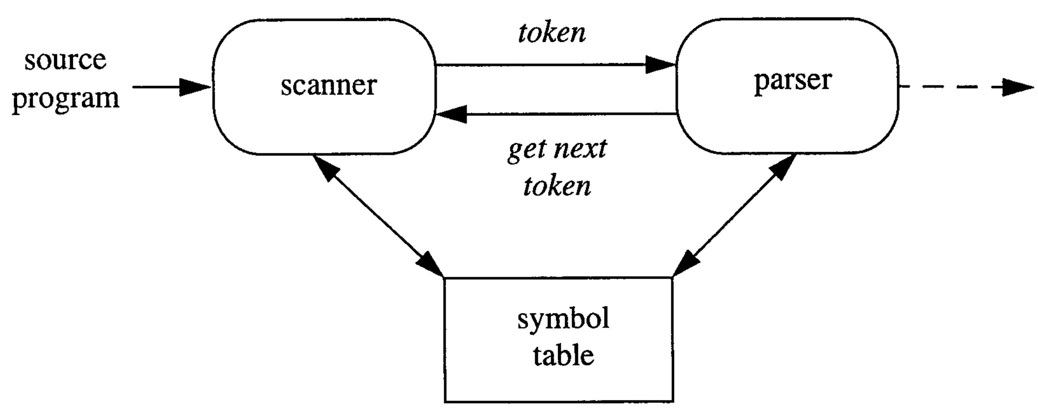


Figure 9. Interaction of scanner with parser.

Since the scanner is the part of the compiler that reads the source text, it is usually tasked with certain secondary duties at the user interface. One such task is stripping out from the source program comments and white space in the form of blank, tab, and newline characters. Another is correlating error messages from the compiler with the source program. For example, the scanner may keep track of the number of newline characters it has seen, so that a line number can be associated with an error message.

a. Token Identification

When talking about lexical analysis, the terms *token, pattern*, and *lexeme* are used with specific meanings. Examples of their use are shown in Figure 10. In general, there is a set of strings in the input for which the same token is produced as output. This set of strings is described by a rule called a *pattern* associated with the token. The pattern is said to *match* each string in the set. The actual notation used to specify a pattern is called a regular expression. A *regular expression* is a simple notation that precisely defines a specified set of character sequences or combinations. A *lexeme* is a sequence of characters in the source program that is matched by the pattern for a token. For example, in the O-ODDL statement

Class Faculty : **Inherit** Person {

the substrings Faculty and Person are lexemes for the token "identifier."

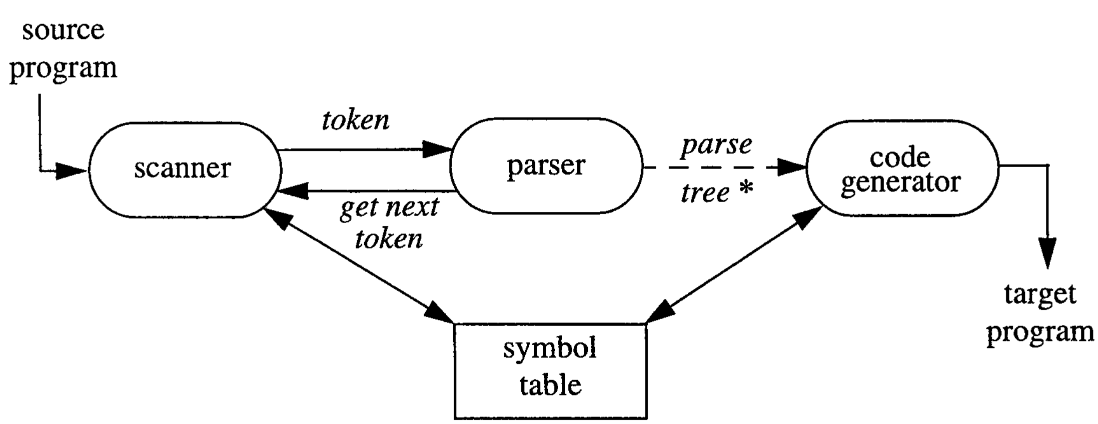
Token	Sample Lexemes	Regular Expression Pattern Description
resrv_word	class, inherit	class, inherit, ... - all reserved words
colon	:	: - only this character
open_brace	{	{ - only this character
id	Faculty, Person	letter+((_(letter | digit))|(letter | digit))* - all non-reserve words that begin with a letter followed by one or more letters or digits

Figure 10. Examples of Tokens

2. The Parser

The parser is the second phase of a compiler. It has two primary tasks. The first, is to obtain a string of tokens from the scanner, as shown in Figure 11, and verify that the string can be generated by the context-free grammar of the source language. A *context-free grammar* describes the precise syntax of a programming language. The second parser task is to simply report any syntax errors in an intelligible fashion.

There are three general types of parsers for grammars: universal parsing methods, top-down, and bottom-up parsing methods. A universal parser is the most powerful, but top-down and bottom-up parser are more efficient. As indicated by their names, top-down build parse trees from the top (root) to the bottom (leaves), while bottom-up parsers start from the leaves and work up to the root. A *parse tree* is a hierarchical structure used in the analysis of the grammatical phrases of a source program, and will be further defined in the section *a.* In both top-down and bottom-up parsers, the input is scanned from left to right, one symbol at a time. We are using a top-down parsing approach for our O-ODDL Compiler.



(*) a *parse tree* is a structured graphical representation of a token string.

Figure 11. Position of the parser in the compiler model.

a. Grammar and Production Rules

As stated earlier, a *context-free grammar* describes the precise syntax of a programming language. And, that syntax allowed in a programming language is specifically delineated by what is called *production rules.* So, production rules are used in describing a context-free grammar. All context-free grammars have the following four components: 1) A set of tokens, known as terminal symbols, e.g., identifiers, reserve_words, and symbols. 2) A set of nonterminals, e.g., statements and expressions. 3) A set of productions where each production consists of a nonterminal, called the *left side* of the production, an arrow, and a sequence of tokens and/or terminals, called the *right side* of the production. 4) A designation of one of the nonterminals as the *start symbol.*

A context-free grammar naturally describes the hierarchical structure of many programming constructs. For example, an if-else statement in the C language has the following form.

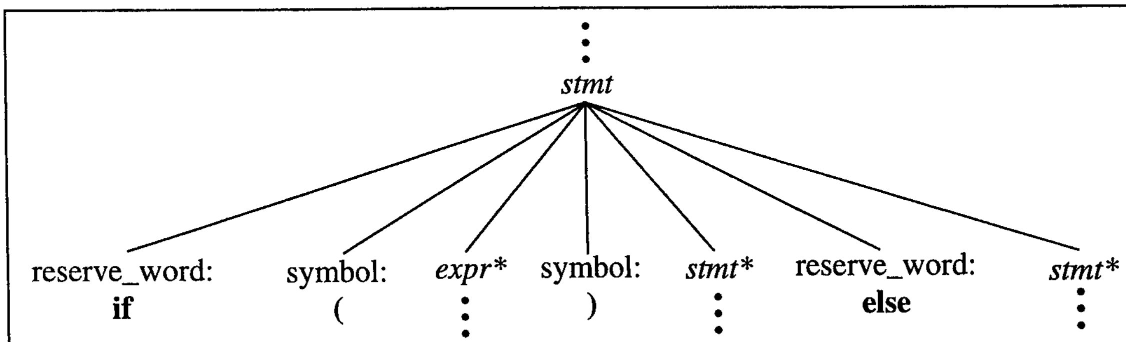
if (expression) statement ***else*** statement

That is, this statement is the concatenation of the reserve_word ***if***, an opening parenthesis, an expression, a closing parenthesis, a statement, the reserve_word ***else***, and another statement. Using the variable *expr* to denote an expression and the variable *stmt* to denote a statement, this structuring rule can be expressed as

stmt → ***if*** *(expr) stmt* ***else*** *stmt*

in which the arrow may be read as "can have the form." Such a statement is an example of a *production rule*.

Only after the context-free grammar and production rules of a program language have been defined, can a syntactic analysis of all feasible language statements be possible. This syntactic analysis of a prospective grammatical phrase is accomplished by using a language's production rules to derive and verify the syntax of that statement. One method to verify syntax is to use parse trees. A *parse tree* is graphical representation of a particular grammatical phrase is derived in a language, where interior nodes correspond to a production rules, and exterior nodes (leaves) correspond to terminal symbols. Figure 12 is an excerpt of a possible parse tree for the if-else statement from above.



(*) *expr* and *stmt* are nonterminals, therefore, they must have corresponding structures that lead to terminals below them.

Figure 12. An example parse tree excerpt.

3. The Code Generator

The final phase of our compiler model is the code generator. It takes as input a parse table representation of the source program produced during the parsing phase, and produces as output an equivalent target program.

The design of a code generator is influenced by several factors. Those factors would normally include issues such as memory management, instruction selection, register allocation, and evaluation orders. But, these issues are only important to a compiler that is intended to produce elaborate programming code. For our compiler model, the only important design issue was the structure of intended output. The structure of the intended output required of our O-ODDL Compiler will be discussed in Chapter VII.

B. LEX AND YACC

Lex and YACC are compiler writing tools. More specifically, Lex is a tool for building lexical analyzers (scanners), hence the name Lex [Ref. 11]. And YACC, which stands for Yet Another Compiler Compiler, is a tool for generating a parser from a list of production rules [Ref. 12].

1. Key Features

Lex takes a set of descriptions of possible tokens generates a C routine. This routine, called *yylex()*, partitions the input stream into specified tokens and communicates these tokens to the parser. The token descriptions that Lex uses are regular expressions, which were discussed earlier in section 1.a.

YACC is a program generator for the syntactic processing of token input streams. The program generated is called *yyparse()*. What YACC requires is a specification of the input language structure (a set of production rules), and the user's code (for target program generation). Once given a set of production rules and user's code, YACC can then generate a program, the parser, that syntactically recognizes the input language and allows invocation of user's code throughout this recognition process. 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. Figure 13 show how Lex and YACC tie in with the compiler model. In this figure, lex.yy.c is the C code produced by Lex, and *yylex()* is the compiled sub-routine program result. Similarly, y.tab.c is the C code produced by YACC, and *yyparse()* is compiled program result [Ref. 13].

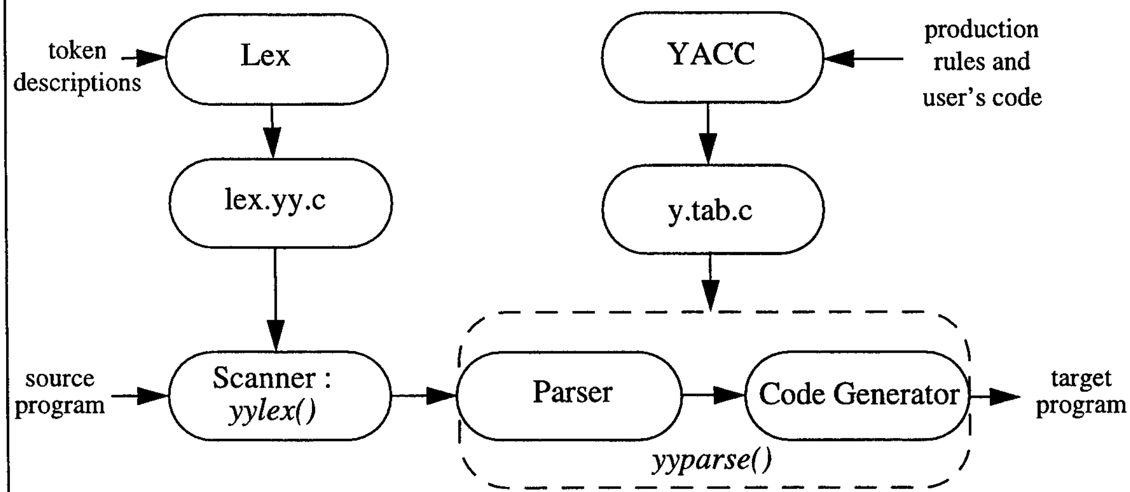


Figure 13. Overall compiler flow diagram.

2. Decision To Use

Our initial foray into production of the O-ODDL Compiler was to write a scanner and parser entirely by hand in the C++ programming language. We had elected to use the C++ language because of its object-oriented properties. But, we later realized that the utilization of this language would inevitably present implementation problems with the existing M^2DBMS.

Our decision to use Lex and YACC was imposed by two project constraints. The first constraints was that the scanner we wrote would be the same scanner that the O-ODML Compiler writing team would ultimately use. But, for reasons discussed in [Ref. 5], they were forced to utilize YACC to write their parser. And YACC, will not accept or use a scanner routine written in the C++ language. The second constraint was that our O-ODDL

Compiler would have to be incorporated into an existing system and interface, the M^2DBMS, that was entirely written in the C language. Some cross language communications between C and C++ are not possible nor allowed.

The next three Chapters will discuss the specific implementation issues involved in producing each of the three respective major compiler components of our O-ODDL Compiler.

V. OBJECT-ORIENTED DDL SCANNER

A. IMPLEMENTATION OVERVIEW

In this Chapter, we will discuss the details involved in building a scanner with Lex, the compiler writing tool. As stated previously, a scanner is the first phase of the compilation process. A scanner takes an arbitrary input stream of characters and *tokenizes* them, i.e., divide up the input stream into lexical tokens. This tokenized output is then used as input for the next phase of the compilation process, the parsing.

The implementation of the scanner in the compilation process is as a subroutine of the parser. Such an executable subroutine is not actually produced by Lex. What Lex actually does produce is a file, named (l*ex.yy.c*). It is this file that produces a C routine called *yylex(),* the actual scanner routine, after compilation with a regular C compiler. It should be noted that we changed the names of *lex.yy.c* and *yylex()* to *lex.ddl.c* and *ddllex()*, respectively, in order to alleviated potential naming conflicts with created files and functions produced by the O-ODML Compiler. So, in order to produce an executable scanner, we used a regular C compiler to compile the lex.ddl.c file. The executable scanner routine was the result of the compilation.

B. SCANNER SPECIFICATION

There were three steps in writing the Lex specification for the scanner component of our O-ODDL compiler. In the first step, we identified the tokens and lexemes that would be recognized in our object-oriented language. In the second step, we specified the patterns in which these tokens could assume with regular expressions. The third and last step was to write the Lex specification in the correct format recognized by Lex.

1. Tokens Recognized in the O-ODDL

We were tasked to produce a scanner that could be jointly used by the O-ODDL and O-ODML Compilers. So, after the data requirements and construct representations of the new O-ODDL and O-ODML Compilers were completed, a complete appreciation of the token requirements could be formed. These tokens included: all the required reserved words, all the character symbols used, including special characters with certain meanings like EOF, which means end_of_file, and all the language variables, i.e., identifier names and numerical strings. The primary task of any scanner is to recognize a specified set of tokens. If a scanner encounters an unspecified token, it should gracefully terminate because this would be considered an error condition. Figure 14 is a complete listing of all the valid

Reserve_Word Tokens

ADD	END	MAX
AND	END_IF	MIN
AVG	END_LOOP	MOD
BEGIN	FIND_MANY	NOT
CHAR	FIND_ONE	NULL
CHAR_STRING	FLOAT	OR
CLASS	FOR	PROJECT
CONTAINS	IF	QUERY
COUNT	IN	READ_INPUT
COVER	INHERIT	SET_OF
DELETE	INSERT	STRING
DISPLAY	INTEGER	THEN
EACH	INVERSE_OF	WHERE
ELSE	IS	

Symbol Tokens

EOF	,	>	[	;
EOL	+	>=	]	<
SPACE	-	=	*	<=
TAB	/	(	//	{
:	/=	)	:=	}

Variable Tokens

identifiers
float_constants
integer_constants
string_constants

Figure 14. A Listing of Valid O-ODDL Compiler Scanner Tokens.

tokens that our scanner was designed to recognize. Any and all other token are then to be considered invalid, and therefore, an error.

2. Valid Token Patterns

The pattern of a token is a precise specification of the set of character strings in which describe a particular token. The only pattern which will describe a reserve_word or symbol token is an accurate copy of the reserve_word or symbol token in question. For example, the only pattern for the reserve_word ADD is exactly the word add. But, note that an accurate copy of a reserve word need not be case specific, because we have designed the scanner to be insensitive to letter case.

The only tokens in which there are numerous character strings would apply are the four variable tokens. Figure 15 is a listing of the variable tokens with their corresponding pattern description using the notation of regular expression.

Variable Tokens	Regular Expression Description
identifier (id)	letter+((_ (letter | digit))|(letter|digit))* | letter+((_ (letter | digit))|(letter|digit)). letter+ ((_ (letter | digit))|(letter|digit))
float_constant (	digit+ (digit+)* . digit+(digit+)*
integer_constant	digit+((digit+)*
string_constant	"printable chars, ASCII 32-126, and TAB"

Key: * .... Means 0 or more
+ .... Means 1 or more
() .... Groups of options, select one.
| .... Separates options
digit .... 0..9
letter .... Means A-Z or a-z
Note: Language is case insensitive.

Figure 15. A Listing of Token Patterns.

3. Lex Implementation

The following is a discussion of the proper format required of a Lex program. Please refer to [Ref. 11] and [Ref. 13] for complete discussions of all the intricacies of this language and its corresponding format. A Lex program consists of three parts as shown in Figure 16: the definition section, the rules section, and the user subroutines. The parts are

```
... definition section ...
%%
... rules section ...
%%
... user subroutines ...
```

Figure 16. The general Lex program format.

separated by lines consisting of two percent signs. the first two parts are required, although a part may be empty. The third part and the proceeding %% line may be omitted [Ref. 13].

The definition section can include definitions, internal table declarations, start conditions, and translations required of the scanner. Lines that start with whitespace are copied verbatim to lex.yy.c, the lex generated C file. The only entries we had in this section were C include declarations for required C library header files.

The rules section contains pattern lines and C code. A line that starts with whitespace, or material enclosed in "%{" and "%}" is C code and is copied verbatim to the generated C file. A line that start with anything else is a pattern line. Pattern lines contain a pattern, i.e. a regular expression if applicable, followed by some whitespace and C code when the input matches the pattern. If the C code spans multiple lines in length, it must be enclosed in braces {}. The final pattern in this section handles the case in which input characters match no specified pattern. In this case, an error condition is raised and outputted to the user somehow. An example of our rules section specification would be

```
add        { return (ADD);}
```

where the word add is the pattern to be matched, and the statement enclosed within the braces is the corresponding C code to be executed upon a successful match.

The contents of the user subroutine section is copied verbatim by Lex to the generated C file. This section typically includes routines called from the rules section. Since this section is completely optional and the fact that our scanner implementation did not require any subroutines, we had no input for this section in our Lex program. A complete listing of the Lex program specification that produced our O-ODDL Compiler scanner is given in Appendix B.

The parser component of the compiler uses the scanner subroutine produced by Lex, called yylex(), to obtain the individual tokens that form grammatically valid token strings. The next Chapter contains a complete discussion of the parser implementation.

VI. OBJECT-ORIENTED DDL PARSER

A. IMPLEMENTATION OVERVIEW

The previous Chapter discussed how Lex is used to produce a scanner. In this Chapter, we turn our attention to producing a parser with YACC. A parser takes the individual tokens produced by the scanner and groups them together logically. These token grouping or relationships must therefore have a certain meanings according the language being parsed. The meaning of these relationships for a particular language is precisely defined by some grammar with corresponding production rules. In short, what a parser ultimately does is, verify that an input program is written to conform to the grammar and production rules of the reference language being used. If the input program does not conform to the specified grammar and production rules, the parser terminates and reports the error.

The implementation of the parser in compilation process is as a subroutine that is called by some controlling program. The actual controlling program and its interface with the M^2DBMS will be discussed in Chapter VIII. The parser subroutine, *yyparse()*, is produced as a result of using a regular C compiler on the YACC generated C files, (*y.tab.c* and *y.tab.h*). Simlilar to the Lex file and function, both yyparse() and y.tab.c were also changed to *ddlparse()* and *ddl.tab.c* to prevent naming conflicts with the OODML Compiler.

A direct result of using YACC to produce our parser was that, the parsing and code generating components of our O-ODDL Compiler were produced in unison, i.e., their functionality was implemented in the resulting *ddlparse()* subroutine. We treated the parser and code generator as two separate components, and therefore implemented them in two separate stages. The first stage was to produce a functionally correct parser with YACC. The second stage was to add the user code, that was introduced in Chapter IV, to the YACC specification in order to produce an appropriate source code. A thorough description of the code generator and its implementation can be found in Chapter VII.

B. PARSER SPECIFICATION

There were two steps in writing the YACC specification for the parser component of our O-ODDL Compiler. The first step was the formal specification of the O-ODDL by means of specifiying complete grammar and corresponding set of production rules. The second step was to put these grammmar and production rules in to a properly formatted YACC specification. This YACC specification produces a functionally correct O-ODDL Compiler parser.

A complete listing of the grammar and production rules that we used to describe the

start	→	*create_table_list* **EOF**
create_table_list	→	*create_table create_table_list_PRIME*
create_table_list_PRIME	→	*create_table_list* | ε
create_table	→	**CLASS** *class_name create_table_PRIME*
create_table_PRIME	→	{ *attribute_list* } ; | *modifier class_name* { *attribute_list* };
modifier	→	: *modifier_PRIME*
modifier_PRIME	→	**INHERIT | COVER**
attribute_list	→	*attribute_declaration attribute_list_PRIME*
attribute_list_PRIME	→	*attribute_declaration attribute_list_PRIME* | ε
attribute_declaration	→	*type attribute_name* ;
type	→	**CHAR** | **CHAR_STRING** | *class_name* | **SET_OF** *class_name* | **INVERSE_OF** *class_name* | **FLOAT** | **INTEGER**
attribute_name	→	**id** *attribute_name_PRIME*
attribute_name_PRIME	→	**[integer_constant]** | ε
class_name	→	**id**

Key: (1) Nonterminals are in italics
(2) RESERVED WORDS ARE IN **BOLD UPPERCASE**
(3) token types are in **bold** lowercase, e.g., **id** and **integer_constant**
(4) ε - stands for the empty case
(5) | - separates possibilities for the same symbol

Figure 17. The O-ODDL Grammar and Production Rules.

O-ODDL is given in Figure 17. Their format is in accordance with requirements outlined in the Grammar and Production Rules section of Chapter IV.

1. YACC Implementation

The following is a discussion of the proper format required of a YACC program specification. Please refer to [Ref. 12] and [Ref. 13] for a more detailed discussion of all the nuances of a YACC specification. A YACC program has the same three-part structure as a lex specification as shown if Figure 18. This is because Lex copied its structure from

```
... definition section ...
%%
... rules section ...
%%
... user subroutines ...
```

Figure 18. The general YACC program format.

YACC. The first section, the definition section, handles control information for the parser. It also generally sets up the execution environment in which the parser will operate. In our YACC specification, we declared all the symbolic tokens that would be used during the O-ODDL Compiler parsing process. The second section contains the rules for the parser, i.e., the reference languages' grammar and production rules. For this section, a complete logical equivalent of all the production rules given in Figure 17 was added. The third and final section is where C code is placed to be copied verbatim into the y.tab.c file, the generated C program. In our specification, this is where we placed a subroutine that was invoked anytime an error condition encountered, called *yyerror()*. What this subroutine does is output the item and corresponding line number of an input program when any parsing error is discovered. A parsing error might include syntax or sematic inconsistencies as per the language specification. In Appendix C, a basic listing of the YACC program specification that produced our basic O-ODDL Compiler parser is given. The only functionality that this basic O-ODDL Compiler parser had was to verify the semantic syntactic correctness of an input program. That is, insure that an input program was written in accordance with the O-ODDL grammar and production rules requirements.

VII. OBJECT-ORIENTED DDL CODE GENERATION

A. IMPLEMENTATION OVERVIEW

The last component of our O-ODDL Compiler is the code generator for producing the target language. The logic behind the code generator is to take the input language, an object-oriented schema specification, and produce the target language, an ABDL schema specification.

The code generator simply stores applicable data from the input language, reformats or reconfigure this data, and produces the target language. The method in which we chose to store the data from the input language was to use linked list data storage structures. The benefit of using linked list data structures are two fold. First, link list data structures are dynamic in that they can vary in size and length depending on the input stream. Having a dynamic memory allocation data structure was a specific requirement for our code generator, because object-oriented database schema specifications can be of varying lengths, therefore requiring storage structures of varying lengths. The second benefit of using a linked list data structure was evident in producing the target program in the proper format, because this task then became a problem of just reading the contents of the linked list structure in the appropriate sequence. A complete discussion of all the linked list component structures we created and used can be found in the next section.

B. THE O-ODDL COMPILER DATA STRUCTURES

The object-oriented data model and language interface was developed for a single user system. However, realizing future system requirements would probably require a multi-user system, we designed our interface with this capability already incorporated. Or more specifically, we modeled our interface after exiting M^2DBMS interfaces which already had this capability. Additionally, our object-oriented database interface utilized appropriate existing generic data structures in the existing M^2DBMS interface, i.e., they already existed as part of the overall M^2DBMS interface. These *generic* data structures support our interface, as well as all others supported by the system.

The new O-ODDL Compiler data structures that we developed to tie into the existing overall M^2DBMS interface had two distinct roles, and therefore were of two distinct types. The first type were used primarily used to store information that would be needed in producing the target data language. The second type were used to in producing the Data Dictionary required of the O-ODML Compiler. A full discussion of the Data Dictionary follows in part 2 of this section. The following data structures and their repective connections are provided in schematic format in Appendix D.

1. Target Language Data Structures

The data structures used to generate the target language originate from the object-oriented database schemas. These schemas consist of data regarding the classes and attributes of an object-oriented database. The first data structure used to maintain data is depicted in Figure 19. This structure represents a union. Hence, it is generic because a user can utilize this structure to support our object-oriented interface as well as the other interfaces. The last field of the dbid_node data structure points to a record that contains information about an object-oriented database.

```
union  dbid_node {
    struct  rel_dbid_node      *dn_rel;
    struct  hie_dbid_node      *dn_hie;
    struct  net_dbid_node      *dn_net;
    struct  dap_db_id_node     *dn_dap;
    struct  obj_dbid_node      *dn_obj;
};
```

Figure 19. The dbid_node Data Structure.

A record of the *obj_dbid_node* type is the structure that contains specific information about a particular object-oriented database. The definition of the *obj_dbid_node* data structure is depicted in Figure 20. The first field is a character array containing the name of the object-oriented database. The next field contains an integer value representing the number of classes in the database. The third, forth and fifth fields excluding the final field are pointers to other records containing information about each class in the database. The rest of the fields excluding the final field are pointers to records containing data dictionary information. The data dictionary data structures will be discussed in the next section. The final field is a pointer to the next object-oriented database schema.

```
struct obj_dbid_node {
    char                    odn_name[DBNLength + 1];
    int                     odn_num_cls;
    struct ocls_node       *odn_first_cls;
    struct ocls_node       *odn_curr_cls;
    struct ocls_node       *odn_hidden_cls;
    struct dict_ocls_node  *odn_first_dict_cls;
    struct dict_ocls_node  *odn_curr_dict_cls;
    struct dict_ocls_node  *odn_hidden_dict_cls;
    struct obj_dbid_node   *odn_next_db;
};
```

Figure 20. The obj_dbid_node Data Structure.

The record *ocls_node* contains information about each class in the database and is depicted in Figure 21. This structure is organized similar to the obj_dbid_node structure. The first field of the record holds the name of the class. The second field holds an integer value for the number of attributes in the class. The third and forth fields are pointers to other records containing information about each attribute contained in a class. The last field contains a pointer to the next class in the database.

```
struct ocls_node {
    char                 ocn_name[ANLength + 1];
    int                  ocn_num_attr;
    struct oattr_node    *ocn_first_attr;
    struct oattr_node    *ocn_curr_attr;
    struct ocls_node     *ocn_next_cls;
};
```

Figure 21. The ocls_node Data Structure.

The final structure used to support the definition of the object-oriented database schema is the *oatt_node* data structure, and it is depicted in Figure 22. The first field is an array which holds the name of the attribute. The second field determines the type. An O-ODDL attribute type can either be a class name (representing a composite attribute), integer, float or character. But, due to an ABDL constraint, the only currently recognized attribute types are integer and string types. The last field contains a pointer to the next attribute in the current class being defined.

```
struct oattr_node {
    char                 oan_name[ANLength + 1];
    char                 oan_type[RNLength + 1];
    struct oattr_node    *oan_next_attr;
};
```

Figure 22. The oattr_node Data Structure.

2. Data Dictionary Data Structures

The reasoning behind having to create a data dictionary for an object-oriented database is simple. Our object-oriented database language is robust in its ability to portray database information. There is more information contained in an object-oriented schema than can be properly and completely conveyed in the ABDL target language translation.

That is, the hierarchical structural information embedded within an object-oriented schema representation can not be represented in the ABDL. Two examples of information that can not be conveyed in an ABDL translation are inheritance and covering property reference information. It is this type of information that the O-ODML Compiler needs to properly format data queries. In short, a data dictionary is persistent record of all the information contained within an object-oriented schema representation.

The data structures used to produce a data dictionary are very similar to those used to produce the target data language. The only differences being the addition or deletion of a few fields to each data structure. As stated above, the data dictionary data structures are "connected" to a particular object-oriented database via the obj_dbid_node record for that database. In Figure 20, the sixth, seventh, and eighth fields are pointers to records with data dictionary information.

The first data structure used to maintain data dictionary data is depicted in Figure 23. The *dict_ocls_node* contains information about each class in the database. The first field of the record holds the name of the class. The second and third fields are pointers to other records containing about each attribute in the class. the last field points to the next class in the database.

```
struct dict_ocls_node {
    char                    ocn_name[ANLength + 1];
    struct dict_attr_node    *dict_first_attr;
    struct dict_attr_node    *dict_curr_attr;
    struct dict_ocls_node    *next_dict_cls;
};
```

Figure 23. The dict_ocls_node Data Structure.

The only other structure used to support the data dictionary is the *dict_attr_node* data structure, and it is depicted in Figure 24. This data structure contains four pieces of information about every attribute: (1) the attribute name, (2) the attribute type, (3) reference class information, and (4) reference relationship type, if applicable.

The first field of a dict_attr_node record is an array which holds the name of the attribute. The second field is an array that contains the attribute type. The two acceptable type that are a result of limitations imposed by what is currently accepted by the ABDL, are: s which stands for character string; and i which stands for integer. The third field is an array that contains the name of a Class in which the current attribute must reference in order to derive some information. The fourth field is an array that contains information on the type of relationship an attribute has with respect to the class named in the third field. Valid relationship are: *inherit*, *cover*, *store*, which short for storage where some specific data item is stored in an alternate more appropriate location, and finally, *asc*, which is short for association. Both the store and asc relationship type are the direct result of the fact that in order to convey the precise meaning of an object-oriented schema specification, hidden "class" data structures had to be created. The two instances in which such a hidden structure were required were in the implementation of the Cover and set_of relationships. For any instance of either of these two relationships, a hidden class must be created that contains relative information on the participating classes, i.e., class OIDs. In Appendix E, a tabular listing of the entire data dictionary that corresponds to the sample FACSTU database can be found. The last field in a dict_attr_node record contains a pointer to the next attribute of the class currently being defined.

```
struct dict_attr_node {
    char                     dict_attr_name[ANLength + 1];
    char                     dict_attr_type[RNLength + 1];
    char                     dict_ref_table[RTLength + 1];
    char                     dict_ref_type[RNLength + 1];
    struct  dict_attr_node   *oan_next_attr;
};
```

Figure 24. The dict_attr_node Data Structure.

C. INTENDED OUTPUT

The output in which the O-ODDL Compiler must generate consists of three items: a template file; a descriptor file; and the data dictionary corresponding to a specific database. All three of these items are automatically generated by the O-ODDL Compiler. The following subsections have complete discussions covering each item.

1. Template File

A *template file* is a specification of the record structure that characterizes the organization of records in a file as recognized in the ABDL, i.e., the record structure format for an attribute-based kernal 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 organized by simply changing the template.

```
Database-name
Number-of-templates
Template-description-1
Template-description-2
        •
        •
        •
Template-description-n
```

Figure 25. The Template File Format.

The template files in the interfaces have a specific structure. The format of a template file for a database with n classes, hence n templates, is shown in Figure 25. A typical template description for a record with m attributes is given in Figure 26. The first field gives the number of attributes in the template. Note that this number are always two more than the number of attributes in the record, i.e., m + 2. This is because the constant attributes, TEMP and OID, are 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, or i representing string, or integer type as per the ABDL restriction stated earlier.

```
Number-of-attributes
template-name
TEMP          s
OID           s
attribute-1   data-type-1
attribute-2   data-type-2
      •
      •
      •
attribute-m   data-type-m
```

Figure 26. A Typical Template Description.

The template file for the Object-Oriented M^2DBMS interface is created by transforming the object-oriented data structure into the template file structure. First, the data structure obj_dbid_node, in Figure 20, is read to get the database name and the number of templates in the database. The number of templates is obtained by totaling the number of class type nodes, ocls_node, that are in the database. The number of attributes corresponding to each class node is obtained by totaling the number of attributes, oattr_node, that are attached to each class node. All these numbers and subsequent class node and attribute node information is obtained by traversing the two linked list structures built with ocls_node and oattr_node data structures. An algorithm for this transformation is presented in Figure 27.

Assertions:

1. The Object-Oriented database O has n class-type nodes $\{C_1, C_2, \ldots, C_n\}$.
2. The Object-Oriented database O has m attribute-type node $\{S_1, S_2, \ldots, S_m\}$.
3. Each class-type node C_i, $i = 1, \ldots$, n, has the class-type name C_i-name.
4. Each attribute-type node S_i, $i = 1, \ldots$, m, has the attribute-type name S_i-name.
5. Each C_i, i = 1, . . . ,n, has T_{Si} attributes.
6. Each attribute $S_{i,j}$, $j = 1, \ldots, T_{Si}$ has the attribute name $S_{i,j}$-name.
7. Each attribute $S_{i,j}$, j = 1, . . . , T_{Si} has the attribute type $S_{i,j}$-type

Algorithm:

```
write Database-name
write Number-of-templates /* i.e., the total number of classes, including internally
                             generated hidden classes */
/* Repeat for each class-type node in database */
for each class-type node C_i in database O do {
    write (T_Si + 1)              /* Number of attributes */
    write C_i-name                /* Class name */
    write "TEMP    s"
    write "TEMP    s"
    /* Repeat for each attribute in the class-type node */
    for each attribute S_i,j in the class-type node C_i do {
        write S_i,j-name    S_i,j-type    /* Attribute name, type */
    }
}
```

Figure 27. Algorithm for Creating the Template File.

2. Descriptor File

While the template file is used to define the record structure of the database, the descriptor file is used to reflect the semantic meanings and intended use of the data. The descriptor file specifies the attributes (or fields) to be regarded as "key" or "indexing" attributes (fields). With the O-ODDL, every attribute in the database can potentially be used as an index. Therefore, all attributes, including internally generated OID attribute fields, are included in the descriptor file.

```
Database-name
TEMP b s
! Name-of-first-class
! Name-of-second-class
   •
   •
   •

! Name-of-last-class
@
Descriptor-definition-1  ┐
Descriptor-definition-2  │
   •                     │ optional
   •                     │ section
   •                     │
                         │
Descriptor-definition-n  ┘
$
```

Figure 28. The Descriptor File Format.

Similar to the template file, the descriptor file also has a specific structure. The format of a descriptor file that has n descriptors is shown in Figure 28. The first entry in the format gives the name of the database. The "TEMP b s" on the second line is a constant that must always be there. Subsequently, for each class in the Object-Oriented database, a line is added with an exclamation mark "!" and a blank space, followed by the class_type name. At the end of the 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 optional descriptor definitions. The $ sign at the last line of the format indicates the end of the descriptor file. The purpose of a *descriptor definition* is to precisely define the range or equality statements that pertain to specific class-types in a database. Since, descriptor definition section of the descriptor file is optional, we elected not implement any for our object-oriented database during this research project.

The algorithm for creating the descriptor file for the implementation of our object-oriented database is given in Figure 29. It is important to note that this algorithm and that of the template file are similar to those for the other interfaces supported by the M^2DBMS.

Assertions:

1. The Object-Oriented database O has n class-type nodes $\{C_1, C_2, \ldots, C_n\}$.
2. Each class-type node C_i, $i = 1, \ldots$, n, has the class-type name C_i-name.

Algorithm:

```
write Database-name
write "TEMP b s"
/* Repeat for each class-type node in database */
for each class-type node Ci in database O do
      write "! " Ci-name
write "@"
write "$"
```

Figure 29. Algorithm for Creating the Descriptor File.

3. Data Dictionary File

As stated previously, the data dictionary provides a persistent record of all the information contained within an object-oriented schema representation. Therefore the data dictionary file contains all the pertinent information described by the object-oriented schema description.

The format of a data dictionary file is shown in Figure 30. It has such a structure so that a sub-routine of the O-ODML Compiler can utilize a reader-subroutine that reads the contents of the file into a linked list data storage structure similar to ours. The first entry in the format gives the name of the database. Next is an at-sign "@". This symbol is at the beginning of every class definition and indicates to the reader-subroutine that another complete class definition follows. The next entry gives the name of the current class being described. A pound-sign "#" immediately follows the class name entry, and this is an indicator for reader-subroutine that four data dictionary attribute elements follow: attribute name, attribute type, reference table, and relation type. Note, for a class name and even some attributes, certain data dictionary attribute elements do not apply, and therefore they remain blank or empty. A sequence of a pound-sign "#" followed by entries for each of

```
Database-name
@
Name-of-class-1
#
class-1-attribute-name
class-1-attribute-type
class-1-reference-table
class-1-relationship-type
#
attribute-name-of-class-1-attribute-1
attribute-type-of-class-1-attribute-1
attribute-reference-table-of-class-1-attribute-1
attribute-relationship-type-of-class-1-attribute-1
        •
        •
        •
#
attribute-name-of-class-1-attribute-m
attribute-type-of-class-1-attribute-m
attribute-reference-table-of-class-1-attribute-m
attribute-relationship-type-of-class-1-attribute-m
        •
        •
        •
@
Name-of-class-n
#
class-n-attribute-name
class-n-attribute-type
class-n-reference-table
class-n-relationship-type
#
attribute-name-of-class-n-attribute-1
attribute-type-of-class-n-attribute-1
attribute-reference-table-of-class-n-attribute-1
attribute-relationship-type-of-class-1-attribute-1
        •
        •
        •
#
attribute-name-of-class-n-attribute-m
attribute-type-of-class-n-attribute-m
attribute-reference-table-of-class-n-attribute-m
attribute-relationship-type-of-class-n-attribute-m
```

Figure 30. The Data Dictionary File Format.

the four attribute elements is entered for every attribute in a class. This entire sequence starting with an at-sign "@" is repeated for every class in an object-oriented schema specification.

D. C CODE IN YACC

As outlined in Chapter VI, the code generating component of our O-ODDL Compiler is created as a result of using YACC to produce our parser. The method by which YACC knows how to implement the code generator is by inserting action C code descriptions for the generation of each of the three required output files. An action C code description is placed in the rules section of a YACC program description immediately following relevant production rules. A complete YACC program listing with code generating capability is given in Appendix F. Additionally, a complete listing of the generated output files for our sample database is given in Appendix G.

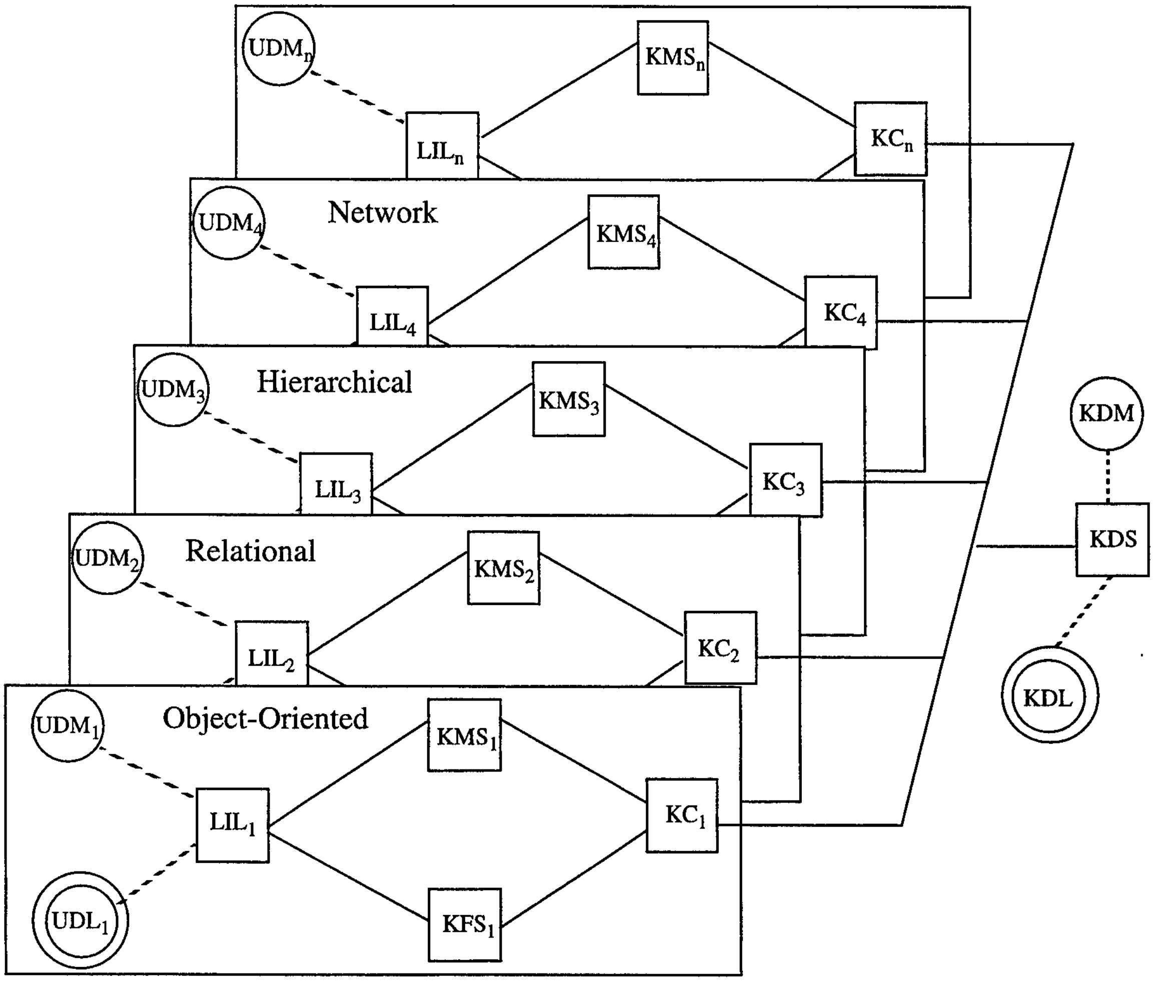
VIII. INCORPATION OF O-ODDL COMPILER INTO EXISTING SYSTEM

Before describing how we incorporated the O-ODDL Compiler into the M^2DBMS, it is important to become familiar with the organization of the M^2DBMS. This overall system organization is utilized by every data language supported by the M^2DBMS. More specifically, the organization and utilization of any supported data model is through virtually identical user interfaces. A pictorial representation of the M^2DBMS with the various user interface modules and their respective control flows is depicted in Figure 31.

A. M^2DBMS EXISTING OVERALL DESIGN AND LOGIC

An original design feature of the M^2DBMS is that it be able to support many data languages. In order to support these data languages, the M^2DBMS requires a separate user interface for each language. All of the user interfaces have identical control flows and structures. The structures that make up every interface are composed of four main modules. As depicted in Figure 31, these modules are the language interface layer (LIL), the kernal mapping system (KMS), the kernal controller (KC) and the kernal formatting system. These four modules comprise the core system for each separate user interface. The kernal database system (KDS) represents the transition system of the kernal data Model/language (KDM/L) and the user data model/language (UDM/L). These components make up the multimodel portion of the multimodel/multilingual database interface and are described individually below.

The LIL routes the user's transaction written UDM/L to the KMS. KMS has two functions. The first identifies whether or not the user is creating a new database. If the user is creating a new database, it transforms the UDM-database definition to the KDM-database definition. This is known as the data-model transformation. Once the KDM-database definition has been established, KMS sends it to KC which in turn routes the KDM-database definition to KDS. The KDS then issues the appropriate commands to the



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

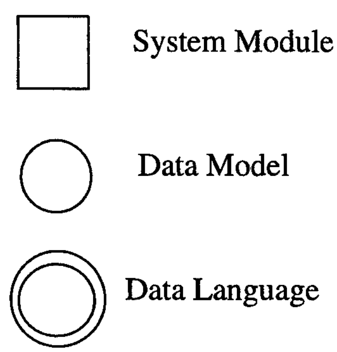


Figure 31. The Multi-model/Multi-lingual Database System.

back-end database supercomputer controller where a new database is created in the KDM form.

The second function of the KMS is the processing of the UDL transaction. In the processing, the KMS translates the UDL transaction into an equivalent KDL transaction. This is known as the data-language translation. The KMS routes the KDL transaction to the KC which then sends the KDL transaction to the KDS for execution. The KC's primary role, in this case, is to oversee the KDL transaction execution.

The KDL transaction is executed in the KDS. Any answer or response is sent to the KC which routes them to the KFS for the KDM-to-UDM transformation. Once the transformation is complete, the KFS routes it to the LIL for the final relay to the user in the user's data model/language form.

Again, the overall language-interface structure consists of the LIL, KMS, KC, and KFS modules, allowing the multimodel/multilingual database system to incorporate different data models and languages. So, each user may create/access a database using his or her data model/language. But, the system stores only one set of data which is in the kernal-data-model form, i.e., in the attribute-based data model.

The actual placement of all O-ODDL components in such a user interface is within the KMS module. The entire contents of the KMS module is pictorially represented in Figure 32. The implementation of all the subcomponents of the KMS module is by means of making each subcomponent a program subroutine. Therefore, the O-ODDL Compiler in essence consists of four subroutines: a scanner subroutine, a parser subroutine, a subroutine that produces the Descriptor and Template Files as a output, and finally, a subroutine that produces a persistent Data Dictionary that can be used by other user interface subcomponents.

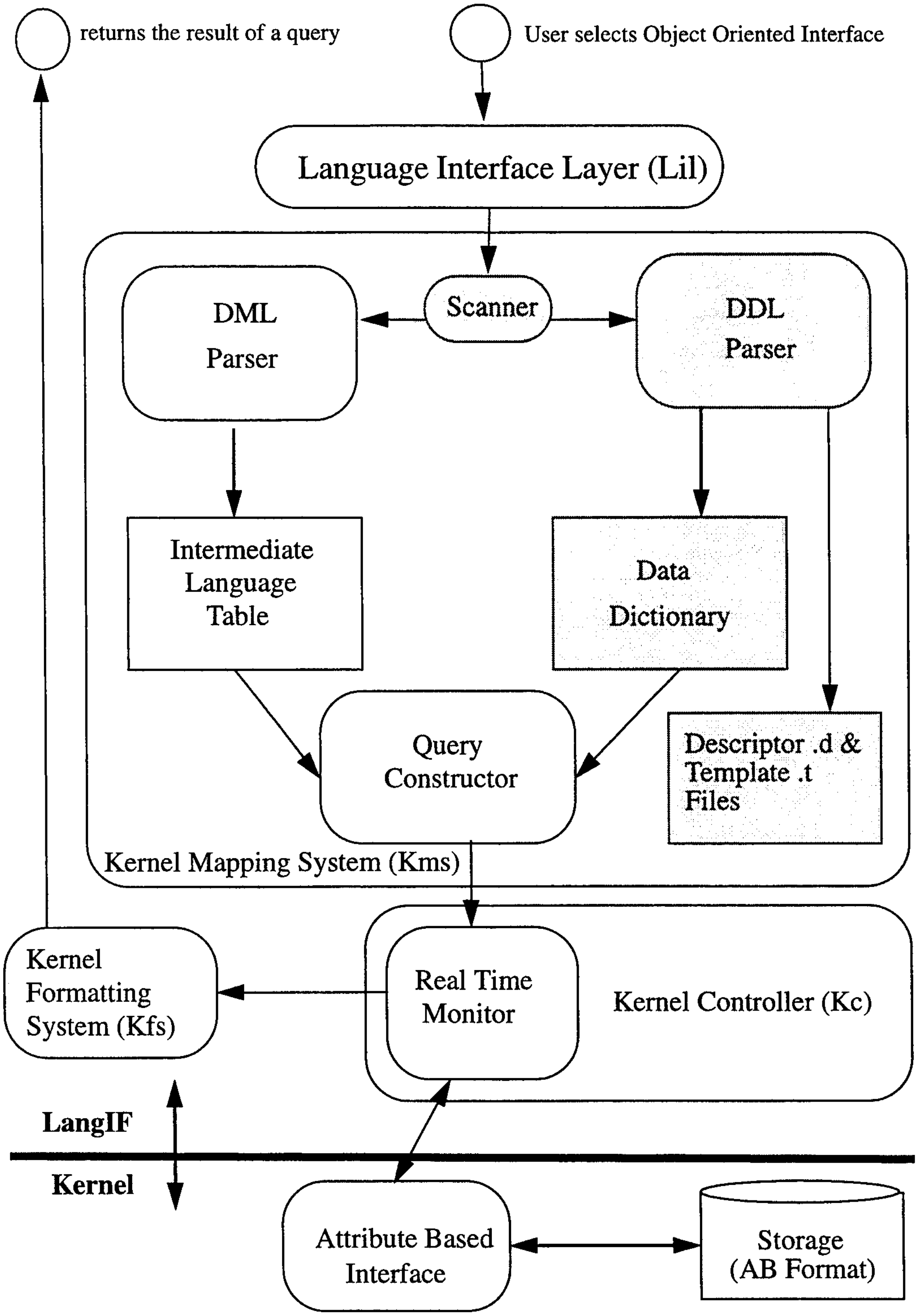


Figure 32. O-ODDL Compiler Component Placement.

B. CONFIGURING DDL COMPILER TO EXISTING DESIGN

The actual merging of the O-ODDL Compiler into the M^2DBMS was just a matter of getting the pertinent subroutines to interface properly within an appropriate M^2DBMS user interface. But, during this incorporation process, we encountered three problems.

1. Problems Encountered

The first problem was encountered during the compilation of the system. The system makes use of the UNIX Makefile tool. With this tool, a single executable file is produced for the execution of the entire system. The problem was that we had initially designed the O-ODDL Compiler to be an entirely self sufficient executable program. But in order to incorporate the O-ODDL Compiler into the system, it must be accessed via program subroutine calls. Not as the execution of an individual program, in which was initially designed.

The second problem also stemmed form the fact that we had initially designed an independent compiler program. The problem was to automatically pass a source file name to the O-ODDL Compiler program subroutine. During the O-ODDL Compiler development, we were simply able to "pipe" a source program name because we had a executable program to reference. But, accessing the O-ODDL Compiler via program subroutine calls do not allow this feature.

The third problem did not present itself until the O-ODML Compiler (see Ref. 5) was incorporated into the system. The designers of the O-ODML Compiler also used the UNIX tools Lex and YACC to produce their compiler. In doing so, the automatic YACC program file naming produced a conflict with our O-ODDL Compiler. That is, Lex and YACC always create program functions and files with the same default name. Examples of such names would include lex.yy.c and yyparse().

2. Problem Solutions

In order to solve the first problem of the O-ODDL Compiler compilation in conjunction with normal system compilation, we had to investigate and learn how the M^2DBMS' compile procedure functions. What we determined was that in order to compile the system, the user first changes to the controlling directory, which in our case was greg/CNTRL/TI. Once in this directory, the user need only execute the %mk command, which initiates a chain reaction compilation of the entire system. This chain reaction is accomplished by successive calls to Makefiles which are contained in every subdirectory that make up the system. A Makefile is simply a programmed set of instructions that are executed one at a time in sequential order. These instructions can include instructions to change to different subdirectories, as well as specific compilations procedures for files contained with that subdirectory. For example, the first instruction in the Makefile contained in the greg/CNTRL/TI/LangIF directory is "cd src/Obj: make". An interpretation of this instruction is to change to the greg/CNTRL/TI/LangIF/src/Obj subdirectory, and then execute the Makefile which resides within the new subdirectory. Therefore, an entire system compilation involves the successive calls to the Makefiles contained within every system subdirectory. The end result of such a compilation in our case is the production of a single executable file, ti.exe, which is located in the greg/CNTRL subdirectory.

So, in order to add the O-ODDL Compiler source files to the system's Kms subdirectory, we followed the same logic as outlined above. First, we copied all the O-ODDL Compiler files into the Kms subdirectory. We then modified the Kms subdirectory Makefile with additional commands to create the requisite object files for the O-ODDL Compiler. Refer to Figure 33 for complete listing of the Kms subdirectory Makefile.

```
# file: Makefile (Obj/Lil)
# path:  db3 /usr/work/mdbs/rich/CNTRL/TI/LangIF/src/Obj/Lil/Makefile
# Insert names of sources here
SRCS= kms.c ddl_compiler.c ddl.tab.c lex.ddl.c $(ALC)alloc.c variable.c dict_functions.c
# Insert names of object files here
OBJECTS= kms.o ddl_compiler.o ddl.tab.o lex.ddl.o $(ALC)alloc.o variable.o dict_functions.o
# Insert names of include files here
INCLUDE= ../../../include
INCLUDES= $(INCLUDE)/licommdata.h $(INCLUDE)/ool.h $(INCLUDE)/ool_lildcl.h
$(INCLUDE)/ooldcl.h
$(INCLUDE)/ddl_functions.h flags.def
ALC= ../Alloc/
CC= cc
#CFLAGS= -g -DEnExFlag -I$(INCLUDE)
CFLAGS= -O -I$(INCLUDE)
LPR= lpr
LPRFLAGS= -p
LIBS= -ll

all: $(INCLUDES) $(OBJECTS)
archive: $(SRCS)
	ci -u $(SRCS)
clean:
	-ci -q $(SRCS)
	-rm -f $(OBJECTS)

print: $(SRCS)
	$(LPR) $(LPRFLAGS) $(SRCS)

lex.ddl.o: ddl_lex.l
	lex ddl_lex.l
	sed -f yy-lsed lex.yy.c > lex.ddl.c
	-rm lex.yy.c
	cc $(CFLAGS) -c lex.ddl.c
ddl.tab.o: ddl_yacc.y
	yacc -d ddl_yacc.y
	sed -f yy-sed y.tab.c > ddl.tab.c
	sed -f yy-sed y.tab.h > ddl.tab.h
	-rm y.tab.c
	-rm y.tab.h
	cc $(CFLAGS) -c ddl.tab.c

variable.o:
	cp $(INCLUDE)/licommdata.h .
	cp $(INCLUDE)/ddl_functions.h .
	cp $(INCLUDE)/ool_lildcl.h .
	cc $(CFLAGS) -c variable.c

HFILES= ddl_functions.h ool_lildcl.h licommdata.h
dict_functions.o:
	cc $(CFLAGS) -c dict_functions.c
	-rm $(HFILES)
$(OBJECTS): $(INCLUDES)
```

Figure 33. The Kms subdirectory Makefile.

The second problem of passing a source files name to the O-ODDL Compiler was solved by the creation of a new function, ddl_compiler(), which resides in the ddl_compiler.c file. In ddl_compiler.c file, we made the following declaration: extern FILE *ddlin. (In reality, Lex produces a default name of yyin, which we changed to ddlin because of the third problem.) By making this declaration, we were then able to assign the source input file name to ddlin. This input file name is entered by the user via the user interface. The user interface prompts the user to enter the source data file name after an appropriate menu selection. Refer to [Ref. 9] for a detailed discussion of the relevant user interface menu selections. The actual compilation of the input file stored in ddlin occurs when the ddl_compiler() function calls yet another function, namely, ddlparse(). The original default name of the ddlparse() function was yyparse(), but it was also changed because of the third problem. In short, the ddl_compiler() function contains all the functionality of the O-ODDL Compiler.

The third and final problem of naming conflicts with other system compilers was solved by using SED, which is yet another UNIX tool. SED is a tool which allows the changing of file names and strings to something more desirable. New names are defined in a SED specification file. For our implementation, we required the following two SED specification files within the Kms subdirectory: yy-lsed and yy-sed.

Before we actually used the SED tool, we created the Lex and YACC files which contained all default names. In the Makefile under the Kms subdirectory (see Figure 33), the line "lex ddl_lex.l" executes the ddl_lex.l file with the Lex tool and produces the lex.yy.c scanner program file. Similarly, the line "yacc -d ddl_yacc.y" executes the ddl_yacc.y file with YACC tool and produces the y.tab.c and y.tab.h parser program files.

After all the O-ODDL program files were created with their default names, we were then able to use the SED tool. The SED specification file yy-lsed was used to change the default names generated by Lex. For example, the program line "sed -f yy-lsed lex.y.c > lex.ddl.c" uses the lex.yy.c file as input and changes every string of this file as specified in the yy-lsed file. The reasoning we used in renaming file and function names was to replace any "yy" prefix with "ddl". For example, the Lex generated yylook() function was renamed ddllook(). Similarly, the program line "sed -f yy-sed y.tab.c > ddl.tab.c and sed -f yy-sed y.tab.h > ddl.tab.h" rename the YACC generated files and functions. As a result of all this renaming of files and functions, any potential conflict between any other compiler is alleviated. For more information regarding the use of multiple Lex and YACC generated compilers on the same system, refer to [Ref. 13].

IX. SUMMARY AND CONCLUSIONS

The work done for this thesis was part of a larger research effort. That larger research effort was to produce an entirely new demonstrable O-ODM and interface for the M^2DBMS. This demonstrable system also included the loading of a sample object-oriented database. It would be this database in which our new O-ODM and O-ODL would subjected to testing in the form of realistic queries, that exercise all system features and capabilities. Our tasking in this research effort was to build the Object-Oriented Data Definition Language Compiler for the system.

In this thesis, we have presented the complete specification and implementation of our Object-Oriented Data Definition Language Compiler. There were three distinct phases in the preparation of this thesis. The first phase was the O-ODM and O-ODL conceptual design. It was in this phase that we defined the specific requirements and capabilities of our new database language. The next phase was the actual building of the O-ODDL Compiler that embodied all the requirements of the first phase. Initially, we wrote our O-ODDL Compiler in the C++ programming language, but were later forced into rewriting the compiler in the C programming language due to constraints imposed by the M^2DBMS. The last phase of our thesis was to incorporate our O-ODDL Compiler into the existing M^2DBMS. Once our O-ODDL Compiler were added the M^2DBMS, we then had to insure that it properly interfaced with all the other components of the object-oriented interface of the M^2DBMS.

We successfully accomplished our task of building an O-ODDL Compiler which properly interfaced with corresponding components of the new object-oriented interface of the M^2DBMS. Our new O-ODDL Compiler implements all the important object-oriented data model's features and constructs. These features and constructs include, but are not limited to, inheritance, class encapsulation, and object reusability.

However, we discovered three limitations during our design process. First, any new interface added to the existing M^2DBMS would have to written in the C programming language, because the kernal language, ABDL, and its corresponding interface were written in C. Second, the ABDL as it is currently implemented does not recognize the float attribute type, i.e., floating point numbers. Finally, our design and utilization of dynamic memory storage structures may be subject to main memory limitations of the computer system being utilized. Each of these limitations is discussed below.

We conclude our thesis with prospects for future research.

A. LIMITATIONS

The three limitations we encountered, the requirement of added programming code to the M^2DBMS must be in the C programming language, lack of recognition of the float attribute type by the ABDL, and a potential main memory limitation, did not hinder our implementation. The requirement of having to implement object-oriented features whilst using a non-object-oriented language (i.e., C vice C++) did force us to change our implementation strategy. Our initial strategy was to use the inherent object-oriented features of the C++ object-oriented programming language. The M^2DBMS could not compile, and therefore was not compatible with C++. Thus, we were forced to the system compatible programming language, C. In fact, using the C programming language proved to be advantageous because of our utilization of the compiler-writing tools, Lex and YACC, which only generate C programming code as output.

The second limitation, non-recognition of the float attribute type by the ABDL, was due to incomplete or erroneous ABDL programming code. However, our overall goal of producing a demonstrable object-oriented interface for the M^2DBMS, was not impeded by this fact. We simply made any O-ODL defined float to be converted into a character string, which could than be recognized by the ABDL.

The third limitation was that of potential system main memory limitations. This limitation arises from the fact that we used dynamic storage structure, i.e., linked lists, in the implementation of our O-ODDL Compiler. The size of main memory occupied by dynamic storage structures is only really limited by the actual size of the main memory. This potential limitation did not manifest itself during the implementation of O-ODDL Compiler, but rather, proved to be a concern during the implementation of the O-ODML Compiler. For additional discussions of this limitation, refer to [Ref. 5]. Our proposed solution to this potential problem was to incorporate a “cleaning” subroutine. This subroutine simply frees allocated memory immediately after a storage structure outlives its usefulness. This solution proved to be adequate for our demonstrable system.

B. FUTURE RESEARCH

There are several issues for future research and they include, but are not limited to, the following: convert the entire M^2DBMS into a more robust object-oriented programming language such as C++; modify the ABDL programming code so that it will recognize the float attribute type; modify the O-ODM and O-ODL to accept multi-class inheritance; modify the object-oriented interface system of the M^2DBMS to accommodate multiple concurrent system users; and finally, build the cross-model links between the object-oriented data model and all other models supported by the M^2DBMS.

The conversion of the entire M^2DBMS into a another programming language would be a major endeavour. But, if the new language chosen were C++, most, if not all, of the code written for the data model interfaces already supported by the M^2DBMS need not change. This is because most C programming code is recognized by C++ program compilers. Another benefit of such a conversion would be that any new future data model interfaces added to the M^2DBMS could be written in C++.

A severe drawback of the ABDL was that it only recognized integer and character string attribute types. The ABDL and its interface was not designed to recognize any other attribute type, including the float attribute type. If anything more than merely a demonstrable system were desired, the ABDL must be modified to recognize floating point numbers.

We designed our O-ODM to handle only single class inheritance. If our O-ODL were required to have the robustness of an object-oriented language like C++, it would most certainly handle multi-class inheritance. But this begs the question, if a database application requires multi-class inheritance, then possibly a data model other than the object-oriented data model might be more appropriate. Multi-class inheritance may certainly be a future possibility, but the issue of an appropriate data model for a particular database application must be explored further.

Our O-ODM interface for the M^2DBMS was designed for a single user in order to expedite the project development. But, the M^2DBMS was initially designed to be a multi-user environment. So, expanding our O-ODM interface to accommodate multiple users would be a natural extension to the system. This would just be the application of an inherent ability of the M^2DBMS and its kernal, the ABDL.

Another inherent ability of the M^2DBMS is the potential for a cross-model capability among all system supported data models. Adding the cross-model links between the functional, hierarchical, network, and relational data models supported by the M^2DBMS, again would be a natural extension of the overall system capabilities. Once these cross-model links were completed, then the object-oriented interface for M^2DBMS system would be complete.

APPENDIX A - SAMPLE OBJECT-ORIENTED (FACSTU)

DATABASE SOURCE CODE

```
class Name{
  char_string  fname;
  char         mi;
  char_string  lname;
};


class Address{
  char_string  street;
  char_string  city;
  char         state[2];
  char_string  zipcode;
};


class Person{
  Name     pname;
  Address  paddress;
  char     sex;
};


class Faculty : inherit Person{
  char_string     dept;
  set_of Course   teaches;          //list courses a faculty member teaches,
};                                  //maps to Course_fac


class Course{
  char_string        cname;
  char_string        cse_no;
  char_string        sec_no;
  Faculty            instructor;   // assigns a faculty member to teach a course
  inverse_of Student.schedule  roster;      // list students enrolled in
};                                          // a course, maps to Student.schule
```

```
class Student : inherit Person{
  char_string      student_no;
  char_string      major;
  set_of Course    schedule;          // list classes a student enrolled
};                                    // in, maps to Course_stu


class Mil_fac : inherit Faculty{
  char_string    rank;
};


class Civ_Fac : inherit Faculty{
  char_string                 title;
  inverse_of Team.advisor    advises;//llist Teams a faculty member advises,
};                                    // maps  to Team.advisor


class Team : cover Student{
  char_string       prjname;
  set_of Civ_Fac    advisor;          // list Civ_fac who are advisors of a team,
};                                    // maps to Team_fac
```

APPENDIX B - THE O-ODDL SCANNER (LEX) PROGRAM LISTING

```
%{
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include "ddl.tab.h"

/*To make the compiler  case-insensitive, lex gets each character as lower case */
#define input() (((yytchar=yysptr>yysbuf?U(*--yysptr) : tolower(getc(yyin)))==
                10?(yylineno++,yytchar):yytchar)==EOF?0:yytchar)
%}
%%
[ \t\n]*              {/* skip whitespace */}
"//".*                ;  /* comment to the end of a line */
add                   { return(ADD);}
and                   { return(AND);}
avg                   { return(AVG);}
begin                 { return(BEGIN_Q);}
char                  { return(CHAR);}
char_string           { return(CHAR_STRING);}
class                 { return(CLASS);}
contains              { return(CONTAINS);}
count                 { return(COUNT);}
cover                 { return(COVER);}
delete                { return(DELETE);}
display               { return(DISPLAY);}
each                  { return(EACH);}
else                  { return(ELSE);}
end                   { return(END_Q);}
end_if                { return(END_IF);}
end_loop              { return(END_LOOP);}
find_many             { return(FIND_MANY);}
find_one              { return(FIND_ONE);}
float                 { return(FLOAT);}
for                   { return(FOR);}
if                    { return(IF);}
in                    { return(IN);}
```

```
inherit              { return(INHERIT);}
insert               { return(INSERT);}
integer              { return(INTEGER);}
inverse_of           { return(INVERSE_OF);}
is                   { return(IS);}
max                  { return(MAX);}
min                  { return(MIN);}
mod                  { return(MOD);}
not                  { return(NOT);}
null                 { return(NULL);}
or                   { return(OR);}
project              { return(PROJECT);}
read_input           { return(READ_INPUT);}
set_of               { return(SET_OF);}
string               { return(STRING);}
then                 { return(THEN);}
query                { return(QUERY);}
where                { return(WHERE);}
\:=                  { return(ASSIGNMENT_OPERATOR);}
[`/='\`<='\`>='\`='\`<'\`>'] { return(RELATION_OPERATOR);}
\[                   { return(OPEN_BRACKET);}
\]                   { return(CLOSE_BRACKET);}
\{                   { return(OPEN_BRACE);}
\}                   { return(CLOSE_BRACE);}
\(                   { return(OPEN_PARENTHESIS);}
\)                   { return(CLOSE_PARENTHESIS);}
\;                   { return(SEMICOLON);}
\,                   { return(COMMA);}
\:                   { return(COLON);}
[*/]                 { return(MULTIPLICATION_OPERATOR);}
[+-]                 { return(ADDITION_OPERATOR);}
\"[^\"]*\"           { yylval.symval = strdup(yytext);
                      return(STRING_CONSTANT);}
[-\+]?[0-9]+[0-9]*   { yylval.symval = strdup(yytext); return
                      (INTEGER_CONSTANT);}
[-\+]?[0-9]+\.?[0-9]* { yylval.symval = strdup(yytext); return
                      (FLOAT_CONSTANT);}
[A-Za-z][A-Za-z0-9]*([_][A-Za-z0-9]+)*(\.[A-Za-z][A-Za-z0-9]*
([_][A-Za-z0-9]+)*)? {yylval.symval = strdup(yytext); return(ID);}
\n                   yylineno++;
.                    printf("invalid character or token encountered at:  %s\n",
                            yytext);
%%
```

APPENDIX C - THE BASIC O-ODDL PARSER (YACC)

PROGRAM LISTING

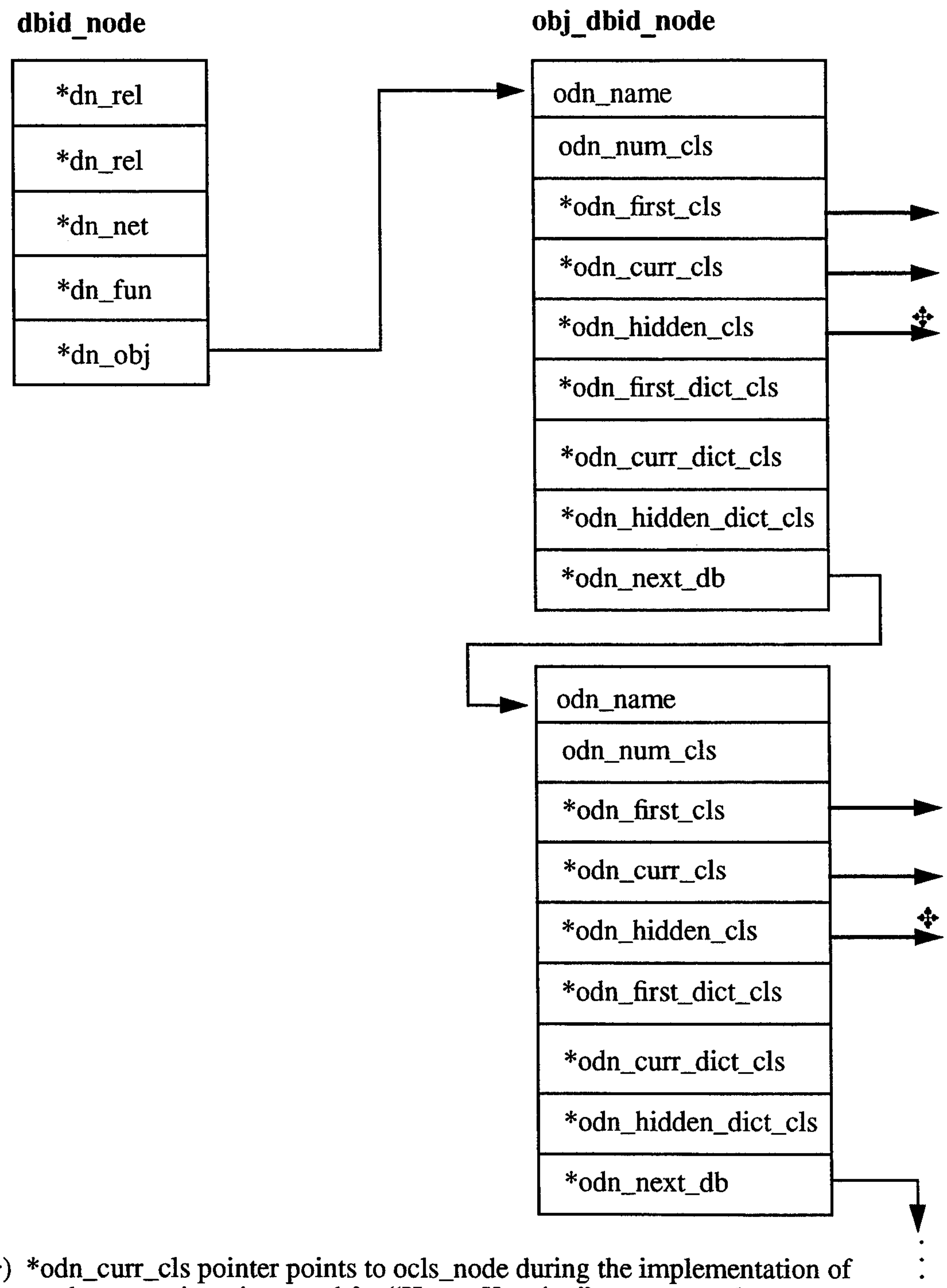
```
%union {
 char t_str[80];
 int t_int;
 }
%token <t_int> ADDITION_OPERATOR
%token <t_int> ASSIGNMENT_OPERATOR
%token <t_int> CLOSE_PARENTHESIS
%token <t_int> COLON
%token <t_int> COMMA
%token <t_int> COMMENT
%token <t_int> DELIMITER
%token <t_int> ILLEGAL
%token <t_int> FLOAT_CONSTANT
%token <t_int> ID
%token <t_int> INTEGER_CONSTANT
%token <t_int> LOGICAL_OPERATOR
%token <t_int> MULTIPLICATION_OPERATOR
%token <t_int> OPEN_PARENTHESIS
%token <t_int> RELATION_OPERATOR
%token <t_int> SEMICOLON
%token <t_int> STRING_CONSTANT
%token <t_int> OPEN_BRACKET
%token <t_int> CLOSE_BRACKET
%token <t_int> OPEN_BRACE
%token <t_int> CLOSE_BRACE
%token <t_int> ADD
%token <t_int> AND
%token <t_int> AVG
%token <t_int> CHAR_STRING
%token <t_int> CHAR
%token <t_int> CLASS
%token <t_int> CONTAINS
%token <t_int> COUNT
%token <t_int> COVER
%token <t_int> DELETE
%token <t_int> DISPLAY
%token <t_int> EACH
```

```
%token <t_int> ELSE
%token <t_int> END_Q
%token <t_int> END_IF
%token <t_int> END_LOOP
%token <t_int> FIND_MANY
%token <t_int> FIND_ONE
%token <t_int> FLOAT
%token <t_int> FOR
%token <t_int> IF
%token <t_int> IN
%token <t_int> INHERIT
%token <t_int> INSERT
%token <t_int> INTEGER
%token <t_int> INVERSE_OF
%token <t_int> IS
%token <t_int> MAX
%token <t_int> MIN
%token <t_int> MOD
%token <t_int> OR
%token <t_int> PROJECT
%token <t_int> READ_INPUT
%token <t_int> SET_OF
%token <t_int> STRING
%token <t_int> THEN
%token <t_int> QUERY
%token <t_int> BEGIN_Q
%token <t_int> NOT
%token <t_int> WHERE
%start start1
%%
start1                      : create_table_list ;

create_table_list           : create_table   create_table_list_prime ;

create_table_list_prime     : create_table_list |  ;

create_table                : CLASS class_name  create_table_prime ;

create_table_prime          : OPEN_BRACE attribute_list CLOSE_BRACE
                              SEMICOLON  |  modifier  class_name
                              OPEN_BRACE attribute_list CLOSE_BRACE
                              SEMICOLON ;
```

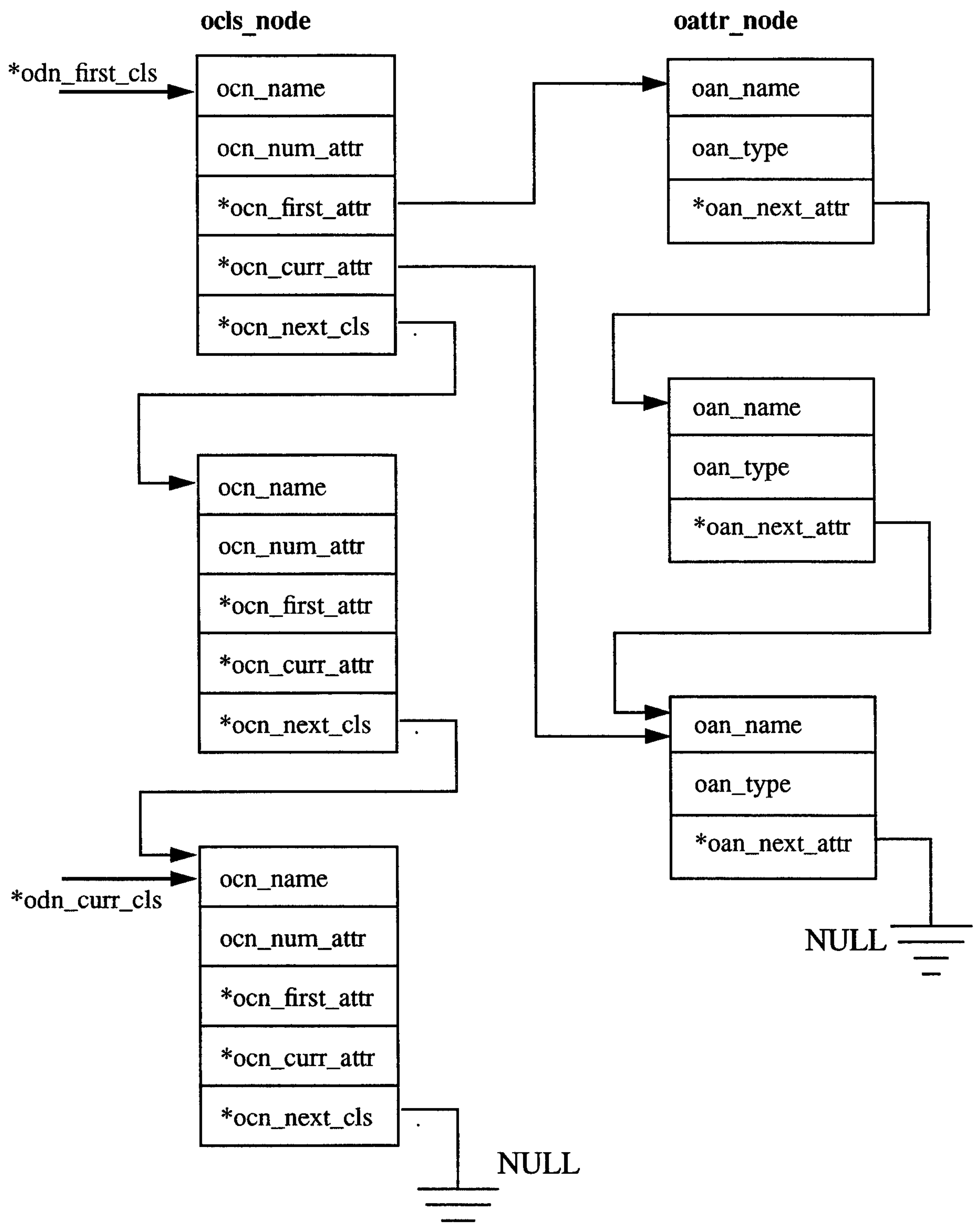
```
modifier                : COLON modifier_prime

modifier_prime          : INHERIT | COVER ;

attribute_list          : attribute_declaration  attribute_list_prime ;

attribute_list_prime    : attribute_declaration attribute_list_prime |  ;

attribute_declaration   : type  attribute_name SEMICOLON ;

type                    : CHAR | CHAR_STRING | class_name | SET_OF
                          class_name  INVERSE_OF class_name  | FLOAT
                          | INTEGER ;

attribute_name          : ID  attribute_name_prime ;

attribute_name_prime    : OPEN_BRACKET INTEGER_CONSTANT
                          CLOSE_BRACKET |  ;

class_name              : ID ;
%%
#include <stdio.h>
extern int yylineno;
void yyerror(s)
char* s;
{
 fflush(stdout);
 fflush(stderr);
 fprintf(stderr, "%s at line %d\n",s,yylineno);
}
main()
{
if (yyparse() == 0)
  {
   fprintf(stderr, "Successfully parsed!!\n");
   exit(1);
  }
else
  {
   printf( "Unsuccessfully parsed!!\n");
   exit(0);
  }
}
```

APPENDIX D - THE OODDL COMPILER DATA STRUCTURES

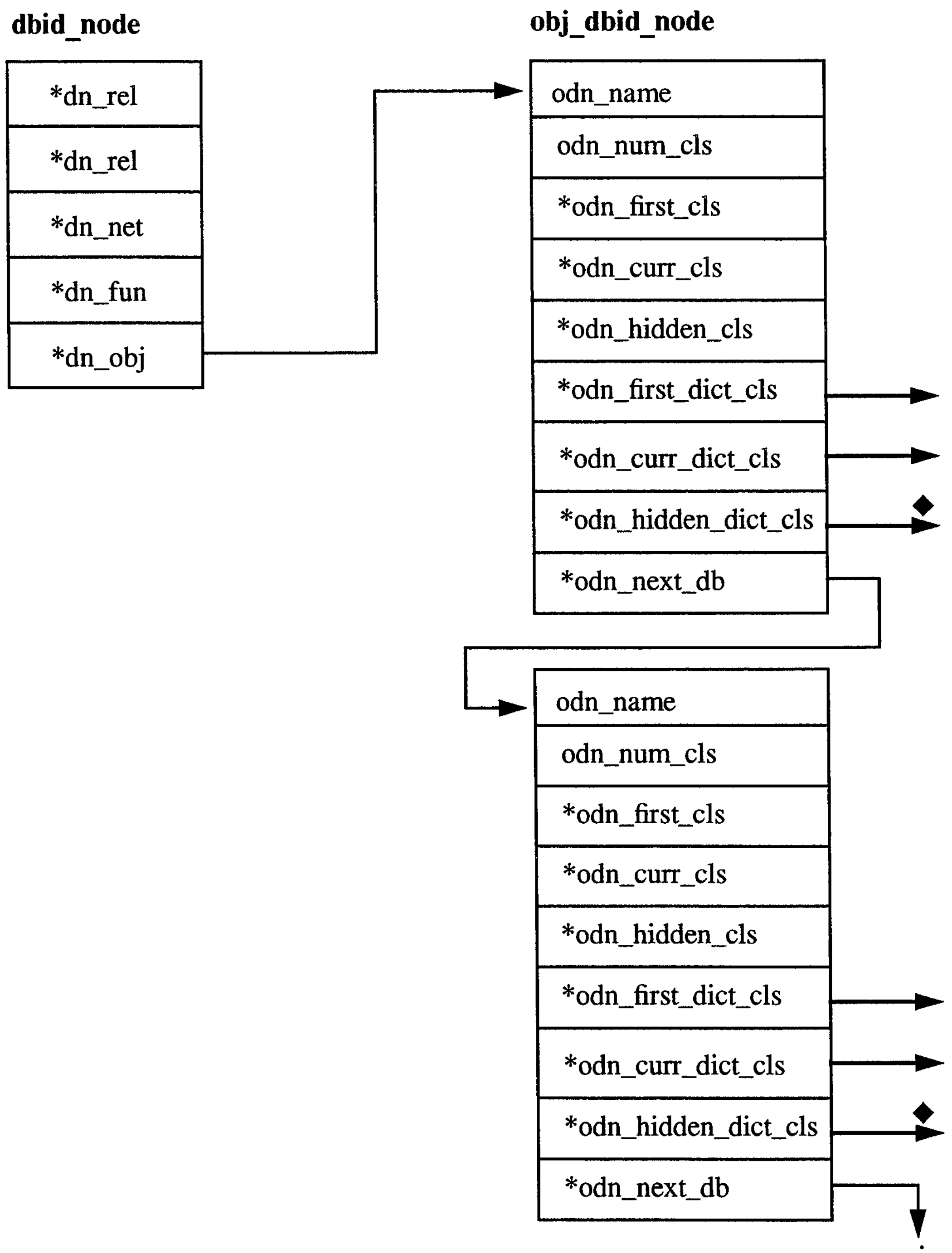
DATA STRUCTURES USED FOR .t AND .d FILE CONSTRUCTION



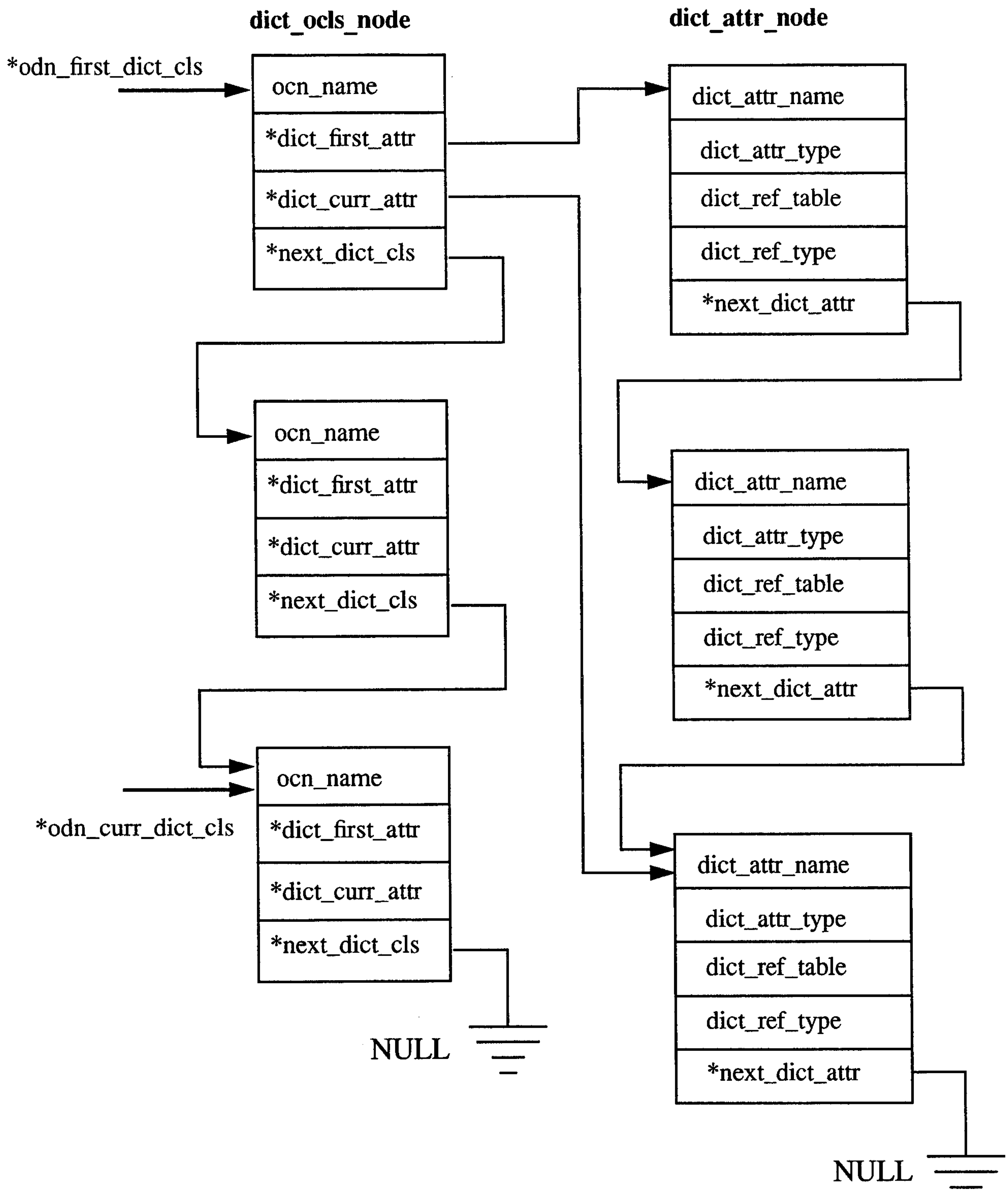
(✣) *odn_curr_cls pointer points to ocls_node during the implementation of code generation. i.e., used for "House Keeping" purposes only.


ocls_node
oattr_node
*odn_first_cls
ocn_name
ocn_num_attr
*ocn_first_attr
*ocn_curr_attr
*ocn_next_cls
oan_name
oan_type
*oan_next_attr
ocn_name
ocn_num_attr
*ocn_first_attr
*ocn_curr_attr
*ocn_next_cls
oan_name
oan_type
*oan_next_attr
oan_name
oan_type
*oan_next_attr
NULL
*odn_curr_cls
ocn_name
ocn_num_attr
*ocn_first_attr
*ocn_curr_attr
*ocn_next_cls
NULL

DATA STRUCTURE USED FOR .dict FILE CONSTRUCTION



(◆) *odn_hidden_dict_cls pointer points to ocls_node during the implementation of code generation. i.e., used for "House Keeping" purposes only.


dict_ocls_node
dict_attr_node
*odn_first_dict_cls
ocn_name
*dict_first_attr
*dict_curr_attr
*next_dict_cls
ocn_name
*dict_first_attr
*dict_curr_attr
*next_dict_cls
ocn_name
*dict_first_attr
*dict_curr_attr
*next_dict_cls
*odn_curr_dict_cls
NULL
dict_attr_name
dict_attr_type
dict_ref_table
dict_ref_type
*next_dict_attr
dict_attr_name
dict_attr_type
dict_ref_table
dict_ref_type
*next_dict_attr
dict_attr_name
dict_attr_type
dict_ref_table
dict_ref_type
*next_dict_attr
NULL

APPENDIX E - THE FACSTU DATA DICTIONARY TABULAR LISTING

CLASS : Name

Name	Attr Type	Ref Table	Rel Type
OID	s		
FNAME	s		
MI	s		
LNAME	s		
NULL			

CLASS: Address

Name	AttrType	Ref Table	Rel Type
OID	s		
STREET	s		
CITY	s		
STATE	s		
ZIPCODE	s		
NULL			

CLASS: Person

Name	Attr Type	Ref Table	Rel Type
OID			
PNAME		Name	ref
PADDRESS		Address	ref
SEX			
NULL			

CLASS: Faculty

Attr Name	Attr Type	Ref Table	Rel Type
OID	s		
OID_PERSON	s	Person	inherit
DEPT	s		
TEACHES	set_of	Course_ faculty	store
NULL			

CLASS: Course

Attr Name	Attr Type	Ref Table	Rel Type
OID	s		
CNAME	s		
CSE_NO	s		
SEC_NO	s		
INSTRUC TOR	s	Faculty	ref
ROSTER	inverse_of	Student.schedule	store
NULL			

CLASS: Student

Attr Name	Attr Type	Ref Table	Rel Type
OID			
OID_PERSON		Person	inherit
STUDENT#			
MAJOR			
SCHEDULE	set_of	Course_ student	store
NULL			

CLASS: Mil_fac

Attr Name	Attr Type	Ref Table	Rel Type
OID	s		
OID_ FACULTY	s	Faculty	inherit
RANK	s		
NULL			

CLASS: Civ_fac

Attr Name	Attr Type	Ref Table	Rel Type
OID	s		
OID_ FACULTY	s	Faculty	inherit
TITLE	s		
ADVISES	inverse_of	Team.advisor	store
NULL			

CLASS: Team

Attr Name	Attr Type	Ref Table	Rel Type
OID	s		
PRJNAME	s		
COVER_ ATTRIBUTE		Student_team	cover
ADVISOR	set_of	Civ_fac_team	store
NULL			

CLASS: Course_faculty

Attr Name	Attr Type	Ref Table	Rel Type
OID	s		
OID_COURSE	s	Course	asc
OID_FACULTY	s	Faculty	asc
NULL			

CLASS: Course_student

Attr Name	Attr Type	Ref Table	Rel Type
OID	s		
OID_COURSE	s	Course	asc
OID_STUDENT	s	Student	asc.
NULL			

CLASS: Civ_fac_team

Attr Name	Attr Type	Ref Table	Rel Type
OID	s		
OID_CIV_FAC	s	Civ_fac	asc
OID_TEAM	s	Team	asc
NULL			

CLASS: Student_team

Attr Name	Attr Type	Ref Table	Rel Type
OID	s		
OID_STUDENT	s	Student	asc
OID_TEAM	s	Team	asc
NULL			

APPENDIX F - THE FINAL O-ODDL PARSER (YACC)

PROGRAM LIST

```
%union {
 char t_str[80];
 int  t_int;
 }
%token <t_int> ADDITION_OPERATOR
%token <t_int> ASSIGNMENT_OPERATOR
%token <t_int> CLOSE_PARENTHESIS
%token <t_int> COLON
%token <t_int> COMMA
%token <t_int> COMMENT
%token <t_int> DELIMITER
%token <t_int> ILLEGAL
%token <t_int> FLOAT_CONSTANT
%token <t_int> ID
%token <t_int> INTEGER_CONSTANT
%token <t_int> LOGICAL_OPERATOR
%token <t_int> MULTIPLICATION_OPERATOR
%token <t_int> OPEN_PARENTHESIS
%token <t_int> RELATION_OPERATOR
%token <t_int> SEMICOLON
%token <t_int> STRING_CONSTANT
%token <t_int> OPEN_BRACKET
%token <t_int> CLOSE_BRACKET
%token <t_int> OPEN_BRACE
%token <t_int> CLOSE_BRACE
%token <t_int> ADD
%token <t_int> AND
%token <t_int> AVG
%token <t_int> CHAR_STRING
%token <t_int> CHAR
%token <t_int> CLASS
%token <t_int> CONTAINS
%token <t_int> COUNT
%token <t_int> COVER
%token <t_int> DELETE
%token <t_int> DISPLAY
%token <t_int> EACH
%token <t_int> ELSE
```

```
%token <t_int> END_Q
%token <t_int> END_IF
%token <t_int> END_LOOP
%token <t_int> FIND_MANY
%token <t_int> FIND_ONE
%token <t_int> FLOAT
%token <t_int> FOR
%token <t_int> IF
%token <t_int> IN
%token <t_int> INHERIT
%token <t_int> INSERT
%token <t_int> INTEGER
%token <t_int> INVERSE_OF
%token <t_int> IS
%token <t_int> MAX
%token <t_int> MIN
%token <t_int> MOD
%token <t_int> OR
%token <t_int> PROJECT
%token <t_int> READ_INPUT
%token <t_int> SET_OF
%token <t_int> STRING
%token <t_int> THEN
%token <t_int> QUERY
%token <t_int> BEGIN_Q
%token <t_int> NOT
%token <t_int> WHERE
%start start1


%%
start1                      : { getGlobalPtr(); }
                                create_table_list ;


create_table_list           : create_table
                              create_table_list_prime ;


create_table_list_prime    : create_table_list | ;
```

```
create_table            : CLASS  {class_flag = 0;
                                  createOclsNode();
                                  createDictClsNode();
                                 }
                          class_name
                          create_table_prime ;


create_table_prime      : OPEN_BRACE
                          attribute_list
                          CLOSE_BRACE
                          SEMICOLON { class_flag = 0;
                                      connectTempCls();
                                    }

                          | modifier
                          class_name
                          OPEN_BRACE
                          attribute_list
                          CLOSE_BRACE
                          SEMICOLON { class_flag = 0;
                                      connectTempCls();
                                      connectTempDictCls();
                                    } ;


modifier                : COLON
                          modifier_prime


modifier_prime          : INHERIT
                          { inherit_flag = 0;
                            createAttrNode(); takeAttrType();
                            createDictAttrNode(); takeDictStringAttrType();
                          }

                          | COVER
                          { cover_flag = 0;
                            createHiddenClass();
                            createHiddenDictClass();
                          } ;
```

```
attribute_list              : attribute_declaration
                              attribute_list_prime ;


attribute_list_prime        : attribute_declaration
                              attribute_list_prime
                              | ;

attribute_declaration       : type
                              attribute_name
                              SEMICOLON { attr_name_flag = 1;} ;


type                        : CHAR
                              { createAttrNode();    takeAttrType();
                                createDictAttrNode(); takeDictStringAttrType();
                              }

                              | CHAR_STRING
                               { createAttrNode(); takeAttrType();
                                 createDictAttrNode(); takeDictStringAttrType();
                               }

                              | { ref_cls_flag = 0;
                                createAttrNode();    takeAttrType();
                                createDictAttrNode(); takeDictStringAttrType();
                               }
                               class_name

                              | SET_OF
                               { set_of_flag = 0; attr_name_flag = 0;
                                createHiddenClass();
                                createHiddenDictClass(); createDictAttrNode();
                               }
                               class_name

                              | INVERSE_OF
                               { attr_name_flag = 0; inverse_of_flag = 0;
                                 createDictAttrNode();
                               }
                               class_name
```

```
                        | FLOAT
                         { createAttrNode(); takeAttrType();
                           createDictAttrNode(); takeDictStringAttrType();
                         }

                        | INTEGER
                         { createAttrNode(); takeIntegerAttrType();
                           createDictAttrNode(); takeDictIntegerAttrType();
                        } ;


attribute_name          : ID
                         { if (attr_name_flag == 1) {
                                  takeAttrName($1);
                                  takeDictAttrName($1);
                            } /*end of if */

                           if (set_of_flag == 0) {
                                  takeDictSeofAttrName($1);
                                  set_of_flag = 1;
                            } /*end of if */

                           if (inverse_of_flag == 0) {
                                  takeDictSeofAttrName($1);
                                  inverse_of_flag = 1;
                            } /*end of if */
                         }
                         attribute_name_prime ;


attribute_name_prime    : OPEN_BRACKET
                         INTEGER_CONSTANT
                         CLOSE_BRACKET
                         | ;


class_name              : ID
                         { if (class_flag == 0){
                                  takeClsName($1);
                                  takeDictClsName($1);
                                  class_flag = 1;
                            } /*end of if */
```

```
                    if (set_of_flag == 0){
                        hiddenClsName($1);
                        hiddenDictClsName($1);
                        dictSetofInfo($1);
                     }  /*end of if */

                    if (cover_flag == 0){
                        hiddenClsName($1);
                        hiddenDictClsName($1);
                        createDictAttrNode();
                        takeDictCoverInfo($1);
                        cover_flag = 1;
                     } /*end of if */

                    if (inherit_flag == 0){
                        takeInheritAttrName($1);
                        takeInheritDictAttrName($1);
                        inherit_flag = 1;
                     }  /*end of if */

                    if (inverse_of_flag == 0){
                        dictInverseofInfo($1);
                     } /*end of if */

                    if (ref_cls_flag == 0){
                        takeDictClsAttrName($1);
                        ref_cls_flag = 1;
                     }  /*end of if */
                   };

%%
#include <stdio.h>
#include <stdlib.h>
extern int ddllineno;

void ddlerror(s)
char* s;

{
 fflush(stdout);
 fflush(stderr);
 fprintf(stderr, "%s at line %d\n", s, ddllineno);
}
```

APPENDIX G - SAMPLE DDL COMPILER OUTPUT FILES

1. FACSTU.d File:

```
FACSTU
TEMP b s
! Name
! Address
! Person
! Faculty
! Course_faculty
! Course
! Student
! Course_student
! Mil_fac
! Civ_fac
! Team
! Student_team
! Civ_fac_team
@
$
```

2. FACSTU.t File:

FACSTU
13

5
Name
TEMP s
OID s
FNAME s
MI s
LNAME s

6
Address
TEMP s
OID s
STREET s
CITY s
STATE s
ZIPCODE s

5
Person
TEMP s
OID s
PNAME s
PADDRESS s
SEX s

4
Faculty
TEMP s
OID s
OID_PERSON s
DEPT s

4
Course_faculty
TEMP s
OID s
OID_COURSE s
OID_FACULTY s

6
Course
TEMP s
OID s
CNAME s
CSE_NO s
SEC_NO s
INSTRUCTOR s

5
Student
TEMP s
OID s
OID_PERSON s
STUDENT_NO s
MAJOR s

4
Course_student
TEMP s
OID s
OID_COURSE s
OID_STUDENT s

4
Mil_fac
TEMP s
OID s
OID_FACULTY s
RANK s

4
Civ_fac
TEMP s
OID s
OID_FACULTY s
TITLE s

3
Team
TEMP s
OID s
PRJNAME s

4
Student_team
TEMP s
OID s
OID_STUDENT s
OID_TEAM s

4
Civ_fac_team
TEMP s
OID s
OID_CIV_FAC s
OID_TEAM s

3. FACSTU.dict File:

```
FACSTU

@
Name
#
OID
s
<space>
<space>

#
FNAME
s
<space>
<space>

#
MI
s
<space>
<space>

#
LNAME
s
<space>
<space>

@
Address
#
OID
s
<space>
<space>

#
STREET
s
<space>
<space>

#
CITY
s
<space>
<space>

#
STATE
s
<space>
<space>

#
ZIPCODE
s
<space>
<space>

@
Person
#
OID
s
<space>
<space>

#
PNAME
s
Name
ref

#
PADDRESS
s
Address
ref

#
SEX
s
<space>
<space>

@
Faculty
#
OID
s
<space>
<space>

#
OID_PERSON
s
Person
inherit

#
DEPT
s
<space>
<space>

#
TEACHES
set_of
Course_faculty
store

@
Course_faculty
#
OID
s
<space>
<space>
```

#
OID_COURSE
s
Course
asc

#
OID_FACULTY
s
Faculty
asc

@
Course
#
OID
s
<space>
<space>

#
CNAME
s
<space>
<space>

#
CSE_NO
s
<space>
<space>

#
SEC_NO
s
<space>
<space>

#
INSTRUCTOR
s
Faculty
ref

#
ROSTER
inverse_of
student.schedule
store

@
Student
#
OID
s
<space>
<space>

#
OID_PERSON
s
Person
inherit

#
STUDENT_NO
s
<space>
<space>

#
MAJOR
s
<space>
<space>

#
SCHEDULE
set_of
Course_student
store

@
Course_student
#
OID
s
<space>
<space>

#
OID_COURSE
s
Course
asc

#
OID_STUDENT
s
Student
asc

@
Mil_fac
#
OID
s
<space>
<space>

#
OID_FACULTY
s
Faculty
inherit

#
RANK
s
<space>
<space>

@
Civ_fac

#
OID
s
<space>
<space>

#
OID_FACULTY
s
Faculty
inherit

#
TITLE
s
<space>
<space>

#
ADVISES
inverse_of
team.advisor
store

@
Team
#
OID
s
<space>
<space>

#
COVER_ATTRIBUTE
<space>
Student_team
cover

#
PRJNAME
s
<space>
<space>

#
ADVISOR
set_of
Civ_fac_team
store

@
Student_team
#
OID
s
<space>
<space>

#
OID_STUDENT
s
Student
asc

#
OID_TEAM
s
Team
asc

@
Civ_fac_team
#
OID
s
<space>
<space>

#
OID_CIV_FAC
s
Civ_fac
asc

#
OID_TEAM
s
Team
asc
$

APPENDIX H - THE COMPILER MANUAL FOR THE OBJECT-ORIENTED DATA DEFINITION LANGUAGE

1. An Introduction:

The OODDL Compiler uses UNIX tools: LEX and YACC. LEX is a scanner tool; YACC is a parser tool. LEX scans the input file which is described in Appendix A. When LEX recognizes a token from the input file, it returns the token to YACC. The OODDL Compiler is a parser-driven compiler; i.e., as the parser, it requests tokens from the scanner one at a time. So, YACC takes all the tokens from LEX and parses them. It checks the sequence of tokens against grammatical rules. If the input satisfies the grammar, YACC gives to the user the message: "Successfully parsed!!". Otherwise, it gives the line number of the line where the error occurs. It also gives the message: "Unsuccessfully parsed!!".

The OODDL Compiler creates the following three files, which are put automatically under the mdbs/UserFiles/ directory. The first file is <database_name>.d file. See Appendix G about it. The second file is <database_name>.t file in Appendix G. The third file is the data dictionary which is <database_name>.dict and can be found also in Appendix G.

2. The Compiler Files:

Unlike before mentioned three files created by the OODDL Compiler, there are files about the compiler itself. Right now, the majority of compiler files are under the mdbs/greg/CNTRL/TI/LangIF/src/Obj/Kms/ directory (See Figure 1 for their display). There are two compiler files which are under the mdbs/greg/CNTRL/TI/Lang/IF/include/ directory (See Figure 2 for their display). There is one file under the mdbs/greg/CNTRL/TI/LangIF/ src/Obj/Alloc/ directory, whose file name is alloc.c. There is another file under the mdbs/ greg/CNTRL/TI/LangIF/src/Obj/Lil/ directory, whose file name is buildddl.c.

ddl_compiler.c	ddl.tab.h	yy-lsed
ddl_lex.l	dict_functions.c	yy-sed
ddl_yacc.y	lex.ddl.c	
ddl.tab.c	template_functions.c	

Figure 1. Files under the Kms directory.

ddl_functions.h	licommdata.h

Figure 2. Files are under the include directory.

3. Description of the Files:

In Figure 1, the file ddl_lex.l is the LEX specification file. It has token definitions. When we run through this file with LEX (i.e.,%lex ddl_lex.l), LEX creates a c file which is the lex.yy.c file. The file lex.yy.c is not in the figure 1, because we changed its name to lex.ddl.c. The reason for this is LEX gives lex.yy.c name as default. There are other implementations in the system, which use LEX and YACC. For example, DML compiler uses LEX and YACC. So, DML compiler's LEX creates lex.yy.c file too. To eliminate confusion, we renamed lex.yy.c file as lex.ddl.c (DML Compiler designers did similar change and renamed their lex.yy.c file as lex.dml.c).

The file ddl_yacc.y is the YACC specification file. It has the grammar rules and function calls from template_fuctions.c and dict_functions.c files. When we run through this file with YACC (i.e.,%yaac -d ddl_yacc.y), YACC creates two files, which are y.tab.c and y.tab.h. Again these file's names are given as default by YACC. With the same reason, which is explained above for lex.yy.c file, we changed these file's names. We renamed y.tab.h as ddl.tab.h and y.tab.c as ddl.tab.c.

The template_functions.c file has functions to create a data structure for .t and .d files. The data structure is a linked list. See Appendix D for the linked list. These functions of the template_functions.c file are called by the ddl_yacc.y file. When YACC matches with a certain grammar rule, it calls the proper function from the template_functions.c file.

The file dict_functions.c has functions to create a data structure for the .dict file. Like template_functions.c functions, these functions create a linked list too. See Appendix D for the linked list. The functions of the dict _functions.c file are called by the ddl_yacc.y file. When YACC matches with a certain grammar rule, it calls the proper function from the dict_functions.c file to create a linked list for the data dictionary.

The file ddl_compiler.c has the main function of the compiler. The function's name is ddl_compiler(). The function ddl_compiler() is invoked from lil.c file. This call activates the OODDL Compiler. Lil.c file is under the mdbs/greg/CNTRL/TI/LangIF/src/Obj/Lil/ directory. The function ddl_compiler() calls ddlparse() function. The ddlparse() is a function which is created by YACC (In fact, YACC creates yyparse() function, but we renamed this function as ddlparse()). When ddlparse() function is invoked compile procedure starts.

The file yy-lsed is SED specification file. It is used to rename the lex.yy.c file and its functions. For more information look Chapter VIII.

The file yy-sed is again SED specification file. It is used to rename the y.tab.c, y.tab.h files and their functions. For more information look Chapter VIII.

In Figure 2, the file ddl_functions.h is a header file. The global variable "dp_ptr" is declared in this file.

The file licommdata.h has data structures for dynamic memory allocation. We took this file from our system. We made some modification in the licommdata.h file.

The file alloc.c has the functions for dynamic memory allocation. These functions allocate dynamically memory for structs, that are declared in the licommdata.h file.

The file buildddl.c has three functions. The first one is o_build_template_file(). This function creates the <database_name>.t file under the mdbs/UserFiles/ directory. The function reads the linked list, that is created during compile time by the template_functions.c functions and writes the requested information into <database_name>.t file.

The second function is o_build_descriptor_file(). This function creates the <database_name>.d under the mdbs/UserFiles/ directory. Like function o_build_template_file(), this function reads the linked list, that is created during compile time by the template_functions.c functions and writes the requested information into <database_name>.d file.

The last function is o_build_dictionary_file(). This function creates the <database_name>.dict file under the mdbs/UserFiles/ directory. This function reads the linked list, that is created during compile time by the dict_functions.c functions and writes the requested information into <database_name>.dict file.

4. How the User Compile and Use the Compiler:

OODDL compiler components are combined with entire system. If the user modifies any OODDL Compiler file, whole system has to be compiled. System uses the UNIX tool Makefile for compile procedure. For more information about compile procedure look Chapter VIII.

To compile the system, the user has to execute the following steps:

a. Login mdbs account.

b. Change the directory to the mdbs/greg/CNTRL/.

c. Execute the line (%rm ti.exe).

d. Change the directory to the mdbs/greg/CNTRL/TI/.

h. Execute the line (%mk). It takes time and does not show anything on the screen.

i. Execute the line (%more make_result). It shows the result of compile. If there is an error, the user has to fix this error and execute the line (%mk) again.

l. Now, system is ready to run. Execute the line (%start).

A note on the use of “<“ and “>”: the name between them is the name of the data base that is given by the user, and it can be changed. In our example, the data base name is FACSTU. So, output files are FACSTU.t, FACSTU.d, and FACSTU.dict.

LIST OF REFERENCES

1. Hsiao, David K. and Kamel, M.N., "The Multimodel and Multilingual Approach to Interoperability of Multidatabase Systems," *International Conference on Interoperability of Multidatabase Systems,* Kyoto, Japan, April 1991.

2. Hsiao, David K., Federated Databases and systems - Part I: A Tutorial on its Data Sharing," *VLDB* Journal, 1, 1992, pp. 127-179.

3. Badgett, Bruce, "The Design and Specification of an Object-Oriented Data Definition Language (O-ODDL)", Master's Thesis, Naval Postgraduate School, Monterey, California, September 1995.

4. Hsiao, David K., "The Object-Oriented Database Management - A Tutorial on its Fundamentals," Proceedings of the Second Far-East Workshop on Future Database Systems, Kyoto, Japan, April 1992.

5. Barbosa, Carlos, M. and Kutlusan, Aykut, "The Design and Implementation of a Compiler for the Object-Oriented Data Manipulation Language (O-ODML Compiler)," Master's Thesis, Naval Postgraduate School, Monterey, California, September 1995.

6. Senocak, Erhan, "The Design and Implementation of a Real-Time Monitor for the Execution of Compiler Object-Oriented Transactions (O-ODDL and O-ODML Monitor)", Master's Thesis, Naval Postgraduate School, Monterey, California, December 1995.

7. Deitel, Harvey M., Deitel, Paul J., *C++: How To Program*, Prentice Hall, 1994.

8. Hsiao, David K., "Interoperable and Multidatabase Solutions for Heterogeneous Databases and Transactions", a speech delivered at **ACM CSC'95**, Nashville, Tennessee, March 1995.

9. Clark, Robert and Yildirim, Necmi, "The Instrumentation of a Kernel DBMS for the Execution of Kernel Transactions Equivalent to their Object-Oriented Transactions," Master's Thesis, Naval Postgraduate School, Monterey, California, September 1995.

10. Aho, Alfred V., Sethi, Ravi, and Ullman, Jeffrey D., *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986.

11. Lesk, M. E. and Schmidt, E., *Lex - A Lexical Analyzer Generator*, Bell Laboratories, Murray Hill, New Jersey, July 1978.

12. Johnson, S. C., *Yacc: Yet Another Compiler-Compiler*, Bell Laboratories, Murray Hill, New Jersey, July 1978.

13. Levine, John R., Mason, Tony, and Brown, Doug, *lex & yacc*, 2nd Edition, O'Reilly & Associates, Inc., October 1992.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center 2
   Cameron Station
   Alexandria, VA 22304-6145

2. Library, Code 52 2
   Naval Postgraduate School
   Monterey, CA 93943-5101

3. Chairman, Code CS 1
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA 93943

4. Professor David K. Hsiao, Code CS/Hs 4
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA 93943

5. Instructor Thomas Wu, Code CS/Wq 4
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA 93943

6. Ms. Doris Mlecsko 2
   Code P22305
   Weapons Division
   Naval Air Warefare Center
   Pt. Mugu, CA 93042-5001

7. Sharon Cain 1
   NAIC/SCDD
   4115 Hebble Creek Road
   Wright Patterson AFB, OH. 45433-5622

8. Deniz Kuvvetteri Komutanligi 2
   Bakanliklar-ANKARA 06600
   Turkey

9. LT Luis M. Ramirez 2
   1091-A Alta Mira Drive
   Santa Clara, CA 95051

10. LTjg Recep Tan 2
Bagarasi Beldesi No:12/B
FOCA-IZMIR 35680
Turkey